Introduction C Programming and Data-Structures

Prof. Pradipta Kumar Mishra

Department Of CSE

Ph.-9938537597

E-Mail:-pradipta.system@gmail.com

Centurion University Technology And Management(CUTM), Bhubaneswar, odisha

Ref.-wiki,e-books and Gate material

Programming in C

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed – since the system and most of the programs that run it are written in C.

- In 1972 Dennis Ritchie at Bell Labs writes C.
- The ANSI standard, or "ANSI C", was released in 1988.

C is popular language because of the following features present in it:

- Compiler Portability
- Standard Library Concepts
- Powerful Operator set
- An elegant syntax
- Usage of hardware using C

C is often called a "Middle Level" programming language where it has capability to access the system's low level functions.

Note:

- Most high-level languages (e.g. Fortran) provides everything the programmer might want to do already built into the language.
- A low level language (e.g. assembler) provides nothing other than access to the machines basic instruction set.

In this C programming study notes, we provide the notes for the following topics along with examples and assignment problems.

- *C basics, Arrays, Pointers, Strings, Functions, Recursion functions*

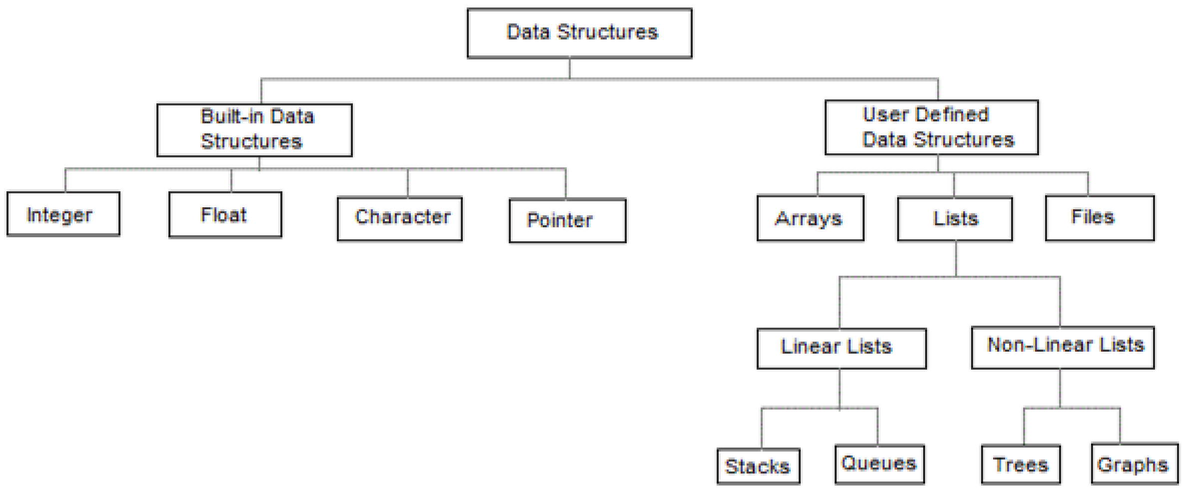
Data Structure

A data structure is a specialised way for organising and storing data in memory, so that one can perform operations on it.

Data structure is all about:

- How to represent data element(s).
- What relationship data elements have among themselves.
- How to access data elements i.e., access methods

Types of Data Structure:



Operations on Data Structures: The operations involve in data structure are as follows.

- **Create:** Used to allocate/reserve memory for the data element(s).
- **Destroy:** This operation deallocate/destroy the memory space assigned to the specified data structure.
- **Selection:** Accessing a particular data within a data structure.
- **Update:** To modify (insertion or deletion) the data in data structure.
- **Searching:** Used to find out the presence of the specified data item in the list of data item.
- **Sorting:** Process of arranging all data items either in ascending or in descending order.
- **Merging:** Process of combining data items of two different sorted lists of data items into a single list.

In this Data Structure, we provide the notes for the following topics along with examples and assignment problems.

- *Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, and graphs.*

Programming in C:

- All C programs must have a function in it called **main**
- Execution starts in function **main**
- C is **case sensitive**
- Comments start with /* and end with */. Comments may span over many lines.
- C is a “free format” language
- All C statements must end in a semicolon (;).
- The **#include <stdio.h>** statement instructs the C compiler to insert the entire contents of file stdio.h in its place and compile the resulting file.

Character Set: The characters that can be used to form words, numbers and expressions depend upon the computer on which the program runs. The characters in C are grouped into the following categories: Letters, Digits, Special characters and White spaces.
C Tokens: The smallest individual units are known as C tokens. C has five types of tokens: Keywords, Identifiers, Constants, Operators, and Special symbols.

- **Keywords:** All keywords are basically the sequences of characters that have one or more fixed meanings. All C keywords must be written in lowercase letters.
    - Example: break, char, int, continue, default, do etc.
- **Identifiers:** A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore ‘_’ followed by zero or more letters, underscores, and digits (0 to 9).
- **Constants:** Fixed values that do not change during the execution of a C program.
    - Example: 100 is integer constant, ‘a’ is character constant, etc.
- **Operators:** Operator is a symbol that tells computer to perform certain mathematical or logical manipulations.
    - Example: Arithmetic operators (+, -, *, /), Logical operators, Bitwise operators, etc.
- **Delimiters / Separators:** These are used to separate constants, variables and statements.
    - Example: comma, semicolon, apostrophes, double quotes, blank space etc.
- **Strings:** String constants are specified in double quotes.
    - Example: “gateexam” is string constant

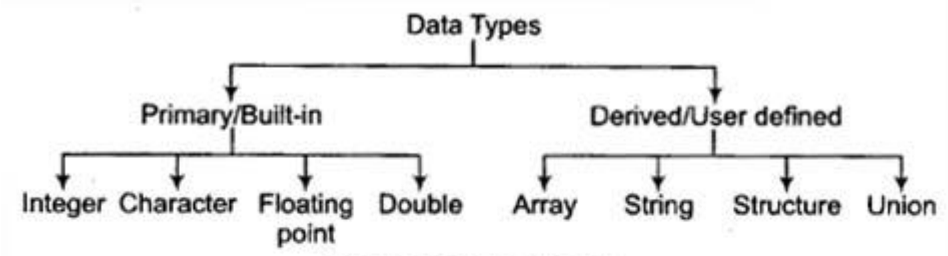
Variable:

- A variable is nothing but a name given to a storage area that our programs can manipulate.
- Each variable in C has a specific type, which determines the size and layout of the variable’s memory.

- The range of values that can be stored within that memory and the set of operations that can be applied to the variable.

Data Types

A data type in a programming language is a set of data with values having predefined characteristics such as integers and characters.



Data types classification

Different Types of Modifier with their Range:

Types of Modifier	Size (bytes)	Range of Values
Int	2	−32768 to + 32767
signed int	2	−32768 to + 32767
unsigned int	2	0 to 65535
short int	2	−32768 to +32767
long int	4	−2147483648 to 2147483647
float	4	−(3.4 E + 48) to + (3.4 E + 48)
double	8	−(1.7 E + 308) to (1.7 E + 308)
char	1	−128 to 127
unsigned char	1	0 to 255

Types of Operators:

- Arithmetic operators (+, -, *, /, %, ++, –)
- Assignment operator (=, +=, -=, *=, etc)
- Relational operators (<, <=, >, >=, !=, ==)
- Logical operators (&&, ||, !)
- Bitwise operators (&, |, ~, ^, <<, >>)
- Special operators (sizeof(), ternary operators)
- Pointer operators (* – Value at address (indirection), & – Address Operator)

Example #1: Arithmetic Operators

```
// C Program to demonstrate the working of arithmetic operators
```

```c
#include <stdio.h>

int main()

{

    int a = 9,b = 4, c;


    c = a+b;

    printf("a+b = %d \n",c);


    c = a-b;

    printf("a-b = %d \n",c);


    c = a*b;

    printf("a*b = %d \n",c);


    c=a/b;

    printf("a/b = %d \n",c);


    c=a%b;

    printf("Remainder when a divided by b = %d \n",c);


    return 0;

}
```

Increment and decrement operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example #2: Increment and Decrement Operators

```c
// C Program to demonstrate the working of increment and decrement operators

#include <stdio.h>

int main()

{

    int a = 10, b = 100;

    float c = 10.5, d = 100.5;


    printf("++a = %d \n", ++a);


    printf("--b = %d \n", --b);


    printf("++c = %f \n", ++c);


    printf("--d = %f \n", --d);


    return 0;

}
```

Output

```
++a = 11

--b = 99

++c = 11.500000

++d = 99.500000
```

Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like **a++** and **a--**. Visit this page to learn more on how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Example #3: Assignment Operators

```c
// C Program to demonstrate the working of assignment operators

#include <stdio.h>

int main()

{

    int a = 5, c;


    c = a;

    printf("c = %d \n", c);


    c += a; // c = c+a

    printf("c = %d \n", c);


    c -= a; // c = c-a

    printf("c = %d \n", c);


    c *= a; // c = c*a

    printf("c = %d \n", c);


    c /= a; // c = c/a

    printf("c = %d \n", c);


    c %= a; // c = c%a

    printf("c = %d \n", c);


    return 0;

}
```

Output

```
c = 5

c = 10

c = 5

c = 25

c = 5

c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Example #4: Relational Operators

```
// C Program to demonstrate the working of arithmetic operators

#include <stdio.h>

int main()

{

    int a = 5, b = 5, c = 10;


    printf("%d == %d = %d \n", a, b, a == b); // true

    printf("%d == %d = %d \n", a, c, a == c); // false


    printf("%d > %d = %d \n", a, b, a > b); //false

    printf("%d > %d = %d \n", a, c, a > c); //false



    printf("%d < %d = %d \n", a, b, a < b); //false

    printf("%d < %d = %d \n", a, c, a < c); //true
```

```c
    printf("%d != %d = %d \n", a, b, a != b); //false

    printf("%d != %d = %d \n", a, c, a != c); //true



    printf("%d >= %d = %d \n", a, b, a >= b); //true

    printf("%d >= %d = %d \n", a, c, a >= c); //false



    printf("%d <= %d = %d \n", a, b, a <= b); //true

    printf("%d <= %d = %d \n", a, c, a <= c); //true


    return 0;


}
```

Output

```
5 == 5 = 1

5 == 10 = 0

5 > 5 = 0

5 > 10 = 0

5 < 5 = 0

5 < 10 = 1

5 != 5 = 0

5 != 10 = 1

5 >= 5 = 1

5 >= 10 = 0

5 <= 5 = 1

5 <= 10 = 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Example #5: Logical Operators

```
// C Program to demonstrate the working of logical operators


#include <stdio.h>

int main()

{

    int a = 5, b = 5, c = 10, result;



    result = (a = b) && (c > b);

    printf("(a = b) && (c > b) equals to %d \n", result);



    result = (a = b) && (c < b);

    printf("(a = b) && (c < b) equals to %d \n", result);



    result = (a = b) || (c < b);

    printf("(a = b) || (c < b) equals to %d \n", result);



    result = (a != b) || (c < b);

    printf("(a != b) || (c < b) equals to %d \n", result);



    result = !(a != b);

    printf("!(a == b) equals to %d \n", result);



    result = !(a == b);

    printf("!(a == b) equals to %d \n", result);
```

```
    return 0;

}
```

Output

```
(a = b) && (c > b) equals to 1

(a = b) && (c < b) equals to 0

(a = b) || (c < b) equals to 1

(a != b) || (c < b) equals to 0

!(a != b) equals to 1

!(a == b) equals to 0
```

Explanation of logical operator program

- **(a = b) && (c > 5)** evaluates to 1 because both operands **(a = b)** and **(c > b)** is 1 (true).
- **(a = b) && (c < b)** evaluates to 0 because operand **(c < b)** is 0 (false).
- **(a = b) || (c < b)** evaluates to 1 because **(a = b)** is 1 (true).
- **(a != b) || (c < b)** evaluates to 0 because both operand **(a != b)** and **(c < b)** are 0 (false).
- **!(a != b)** evaluates to 1 because operand **(a != b)** is 0 (false). Hence, !(a != b) is 1 (true).
- **!(a == b)** evaluates to 0 because **(a == b)** is 1 (true). Hence, **!(a == b)** is 0 (false).

Bitwise Operators

In processor, mathematical operations like: addition, subtraction, addition and division are done in bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

int a, c = 5, d;

The sizeof operator

The **sizeof** is an unary operator which returns the size of data (constant, variables, array, structure etc).

Example #6: sizeof Operator

```
#include <stdio.h>

int main()

{

    int a, e[10];

    float b;

    double c;

    char d;

    printf("Size of int=%lu bytes\n",sizeof(a));

    printf("Size of float=%lu bytes\n",sizeof(b));

    printf("Size of double=%lu bytes\n",sizeof(c));

    printf("Size of char=%lu byte\n",sizeof(d));

    printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));

    return 0;

}
```

Output

```
Size of int = 4 bytes

Size of float = 4 bytes

Size of double = 8 bytes

Size of char = 1 byte

Size of integer type array having 10 elements = 40 bytes
```

C Ternary Operator (?:)

A conditional operator is a ternary operator, that is, it works on 3 operands.

Conditional Operator Syntax

```
conditionalExpression ? expression1 : expression2
```

The conditional operator works as follows:

- The first expression conditionalExpression is evaluated at first. This expression evaluates to 1 if it's and evaluates to 0 if it's false.
- If **conditionalExpression** is true, **expression1** is evaluated.
- If **conditionalExpression** is false, **expression2** is evaluated.

Example #6: C conditional Operator

```
#include <stdio.h>

int main(){

   char February;

   int days;

   printf("If this year is leap year, enter 1. If not enter any integer: ");

   scanf("%c",&February);


   // If test condition (February == 'l') is true, days equal to 29.

   // If test condition (February =='l') is false, days equal to 28.

   days = (February == '1') ? 29 : 28;


   printf("Number of days in February = %d",days);

   return 0;

}
```

Output

```
If this year is leap year, enter 1. If not enter any integer: 1

Number of days in February = 29
```

Conversions

- **Implicit Type Conversion:** There are certain **cases** in which data will get automatically converted from one type to another:
  - When data is being stored in a variable, if the data being stored does not match the type of the variable.

        - The data being stored will be converted to match the type of the storage variable.
        - When an operation is being performed on data of two different types. The “smaller” data type will be converted to match the “larger” type.
            - The following example converts the value of x to a double precision value before performing the division. Note that if the 3.0 were changed to a simple 3, then integer division would be performed, losing any fractional values in the result.
                - average = x / 3.0;
        - When data is passed to or returned from functions.
- **Explicit Type Conversion:** Data may also be expressly converted, using the typecast operator.
    - The following example converts the value of x to a double precision value before performing the division. (y will then be implicitly promoted, following the guidelines listed above.)
        - average = (double) x / y;
        - Note that x itself is unaffected by this conversion.

Expression:

- **lvalue:**
    - Expressions that refer to a memory location are called “lvalue” expressions.
    - An lvalue may appear as either the left-hand or right-hand side of an assignment.
    - Variables are lvalues and so they may appear on the left-hand side of an assignment
- **rvalue:**
    - The term rvalue refers to a data value that is stored at some address in memory.
    - An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.
    - Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side.

C Flow Control Statements

Control statement is one of the instructions, statements or group of statement in a programming language which determines the sequence of execution of other instructions or statements. C provides two styles of flow controls.

1. Branching (deciding what action to take)

2. Looping (deciding how many times to take a certain action)

C if statement

```
if (testExpression)

{

   // statements

}
```

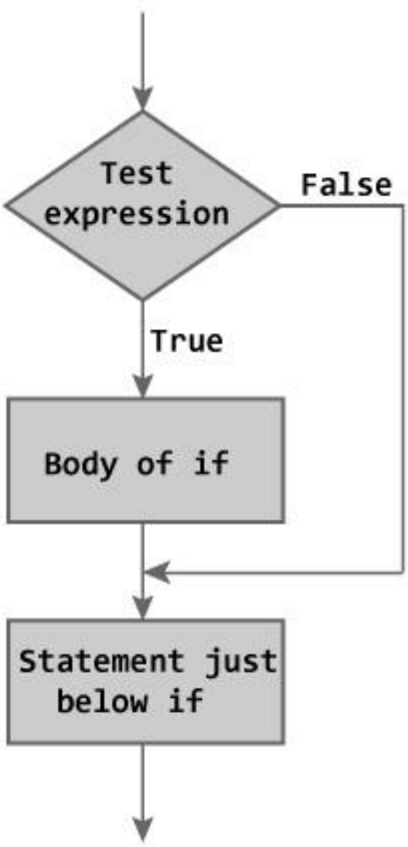
3. The **if** statement evaluates the test expression inside parenthesis.

4. If test expression is evaluated to true (nonzero), statements inside the body of **if** is executed.

5. If test expression is evaluated to false (0), statements inside the body of **if** is skipped.

6. To learn more on test expression (when test expression is evaluated to nonzero (true) and 0 (false)), check out relational and logical operators.

7. Flowchart of if statement



8. Figure: Flowchart of if Statement

Example #1: C if statement

```
// Program to display a number if user enters negative number

// If user enters positive number, that number won't be displayed


#include <stdio.h>

int main()

{

    int number;


    printf("Enter an integer: ");

    scanf("%d", &number);
```

```
    // Test expression is true if number is less than 0

    if (number < 0)

    {

        printf("You entered %d.\n", number);

    }


    printf("The if statement is easy.");


    return 0;

}
```

Output 1

```
Enter an integer: -2

You entered -2.

The if statement is easy.
```

9. When user enters -2, the test expression **(number < 0)** becomes true. Hence, `You entered -2` is displayed on the screen.

Output 2

```
Enter an integer: 5

The if statement in C programming is easy.
```

10. When user enters 5, the test expression **(number < 0)** becomes false and the statement inside the body of **if** is skipped.

11. C if...else statement

12. The **if...else** statement executes some code if the test expression is true (nonzero) and some other code if the test expression is false (0).

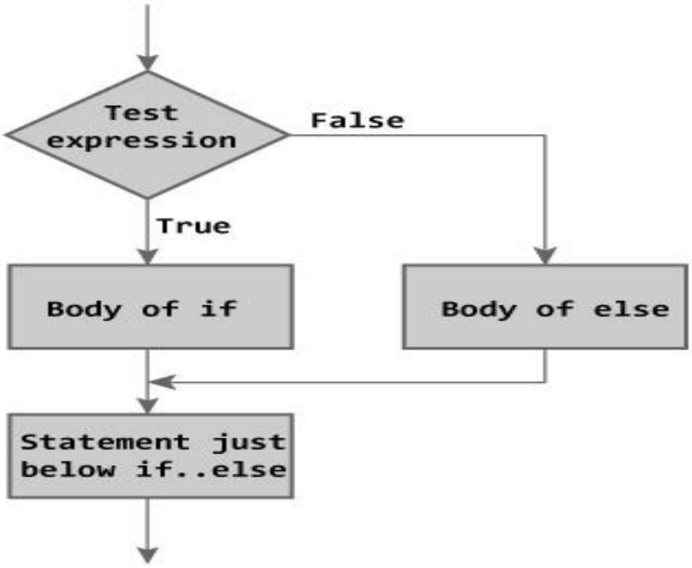
Syntax of if...else

```
if (testExpression) {

    // codes inside the body of if

}

else {

    // codes inside the body of else

}
```

13. If test expression is true, code inside the body of **if** statement is executed; and code inside the body of **else** statement is skipped.

14. If test expression is false, code inside the body of **else** statement is executed; and code inside the body of **if** statement is skipped.

15. Flowchart of if...else statement



16. Figure: Flowchart of if...else Statement

Example #2: C if...else statement

```
// Program to check whether an integer entered by the user is odd or even


#include <stdio.h>
```

```
int main()

{

    int number;

    printf("Enter an integer: ");

    scanf("%d",&number);


    // True if remainder is 0

    if( number%2 == 0 )

        printf("%d is an even integer.",number);

    else

        printf("%d is an odd integer.",number);

    return 0;

}
```

Output

```
Enter an integer: 7

7 is an odd integer.
```

17. When user enters 7, the test expression **(number%2 == 0)** is evaluated to false. Hence, the statement inside the body of **else** statement **printf("%d is an odd integer");** is executed and the statement inside the body of **if** is skipped.

18. Nested if...else statement (if...elseif....else Statement)

19. The **if...else** statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

20. The nested if...else statement allows you to check for multiple test expressions and execute different codes for more than two conditions.

Syntax of nested if...else statement.

```
if (testExpression1)
```

```
{

   // statements to be executed if testExpression1 is true

}

else if(testExpression2)

{

   // statements to be executed if testExpression1 is false and testExpression2 is true

}

else if (testExpression 3)

{

   // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true

}

.

.

else

{

   // statements to be executed if all test expressions are false

}
```

Example #3: C nested if...else statement

```
// Program to relate two integers using =, > or <


#include <stdio.h>

int main()

{

    int number1, number2;

    printf("Enter two integers: ");

    scanf("%d %d", &number1, &number2);
```

```
    //checks if two integers are equal.

    if(number1 == number2)

    {

        printf("Result: %d = %d",number1,number2);

    }


    //checks if number1 is greater than number2.

    else if (number1 > number2)

    {

        printf("Result: %d > %d", number1, number2);

    }


    // if both test expression is false

    else

    {

        printf("Result: %d < %d",number1, number2);

    }


    return 0;

}
```

Output

```
Enter two integers: 12

23

Result: 12 < 23
```

21. You can also use switch statement to make decision between multiple possibilites.

The switch Statement: The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed:

switch (control variable)

{

case constant-1: statement(s); break;

case constant-2: statement(s); break;

:

case constant-n: statement(s); break;

default: statement(s);

}

The Conditional Operators (? :): The ? : operators are just like an if-else statement except that because it is an operator we can use it within expressions. ? : are a ternary operators in that it takes three values. They are the only ternary operator in C language.

```
flag = (x < 0) ? 0 : 1;
```

This conditional statement can be evaluated as following with equivalent if else statement.

```
if (x < 0) flag = 0;
else flag = 1;
```

Loop Control Structure : Loops provide a way to repeat commands and control. This involves repeating some portion of the program either a specified numbers of times until a particular condition is satisfied.

- **while Loop:**

```
initialize loop counter;
while (test loop counter using a condition/expression)
{
    <Statement1>
    <Statement2>
     ...
     < decrement/increment loop counter>
}
```

- **for Loop:**

```
for (initialize counter; test counter; increment/decrement counter)
{
      <Statement1>
```

<Statement2>

…

}

- **do while Loop:**

initialize loop counter;

do

{

<Statement1>

<Statement2>

…

}

while (this condition is true);

while loop

```
// Program to find factorial of a number

// For a positive integer n, factorial = 1*2*3...n


#include <stdio.h>

int main()

{

    int number;

    long long factorial;


    printf("Enter an integer: ");

    scanf("%d",&number);


    factorial = 1;


    // loop terminates when number is less than or equal to 0

    while (number > 0)

    {
```

```c
        factorial *= number;  // factorial = factorial*number;

        --number;

    }



    printf("Factorial= %lld", factorial);



    return 0;

}
```

while loop

```c
// Program to find factorial of a number

// For a positive integer n, factorial = 1*2*3...n



#include <stdio.h>

int main()

{

    int number;

    long long factorial;



    printf("Enter an integer: ");

    scanf("%d",&number);



    factorial = 1;



    // loop terminates when number is less than or equal to 0

    while (number > 0)

    {

        factorial *= number;  // factorial = factorial*number;
```

```
        --number;
    }

    printf("Factorial= %lld", factorial);

    return 0;
}
```

Output

```
Enter an integer: 5
```

Example: Print numbers from 1 to 5 using for loop

```
#include <stdio.h>

int main()
{
int counter;
for(counter=1; counter <= 5; counter++) //loop 5 times
{  printf("%d ", counter);  }
return 0;
}
```

- **The break Statement:** The break statement is used to jump out of a loop instantly, without waiting to get back to the conditional test.

break statement

```
// Program to calculate the sum of maximum of 10 numbers
// Calculates sum until user enters positive number
```

```c
# include <stdio.h>

int main()

{

    int i;

    double number, sum = 0.0;


    for(i=1; i <= 10; ++i)

    {

        printf("Enter a n%d: ",i);

        scanf("%lf",&number);


        // If user enters negative number, loop is terminated

        if(number < 0.0)

        {

            break;

        }


        sum += number; // sum = sum + number;

    }


    printf("Sum = %.2lf",sum);


    return 0;

}
```

- **The continue Statement:** The ‘continue’ statement is used to take the control to the beginning of the loop, by passing the statement inside the loop, which have not yet been executed.

continue statement

```c
// Program to calculate sum of maximum of 10 numbers

// Negative numbers are skipped from calculation


# include <stdio.h>

int main()

{

    int i;

    double number, sum = 0.0;


    for(i=1; i <= 10; ++i)

    {

        printf("Enter a n%d: ",i);

        scanf("%lf",&number);


        // If user enters negative number, loop is terminated

        if(number < 0.0)

        {

            continue;

        }


        sum += number; // sum = sum + number;

    }


    printf("Sum = %.2lf",sum);


    return 0;

}
```

- **goto Statement:** C supports an unconditional control statement, goto, to transfer the control from one point to another in a C program.

C Variable Types

A variable is just a named area of storage that can hold a single value. There are two main variable types: Local variable and Global variable.

- **Local Variable:** Scope of a local variable is confined within the block or function, where it is defined.
- **Global Variable:** Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it.

Global variables are initialized automatically by the system when we define them. If same variable name is being used for global and local variables, then local variable takes preference in its scope.

Storage Classes in C

- A variable name identifies some physical location within the computer, where the string of bits representing the variable's value, is stored.
- There are basically two kinds of locations in a computer, where such a value maybe kept: Memory and CPU registers.
- It is the variable's storage class that determines in which of the above two types of locations, the value should be stored.

We have four types of storage classes in C: Auto, Register, Static and Extern.

1. Auto Storage Class: Features of this class are given below.

- **Storage Location** Memory
- **Default Initial Value** Garbage value
- **Scope** Local to the block in which the variable is defined.
- **Life** Till the control remains within the block in which variable is defined.

Auto is the default storage class for all local variables.

2. Register Storage Class: Register is used to define local variables that should be stored in a register instead of RAM. Register should only be used for variables that require quick access such as counters. Features of register storage class are given below.

- **Storage Location** CPU register
- **Default Initial Value** Garbage value
- **Scope** Local to the block il) which variable is defined.
- **Life** Till the control remains within the block in which the variable is defined.

3. Static Storage Class: Static is the default storage class for global variables. Features of static storage class are given below.

- **Storage Location** Memory
- **Default Initial Value** Zero
- **Scope** Local to the block in which the variable is defined. In case of global variable, the scope will be through out the program.
- **Life** Value of variable persists between different function calls.

4. Extern Storage Class: Extern is used of give a reference of a global variable that is variable to all the program files. When we use extern, the variable can't be initialized as all it does, is to point the variable name at a storage location that has been previously defined.

- **Storage Location:** Memory
- **Default Initial Value:** Zero
- **Scope:** Global
- **Life:** As long as the program's execution does not come to an end.

Operator Precedence Relations: Operator precedence relations are given below from highest to lowest order:

() [] -> .	function calls and indexing
! ~ - (unary) * (unary) &(unary) ++ -- (*type*) sizeof	unary operators (associate right-to-left)
* (binary) / %	multiplication and division
+ (binary) - (binary)	addition and subtraction
<< >>	shifts
< <= >= >	inequalities
== !=	equality
& (binary)	bitwise AND
^	bitwise XOR
|	bitwise OR
&&	logical AND
||	logical OR
?:	ternary if (associates right-to-left)
= += -= *= /= %= &= ^= |= <<= >>=	assignment (associate right-to-left)
,	comma

Example: Displaying a character 'A' and its ASCII value.

```
#include <stdio.h>

int main()

{
```

```
char first_letter;

first_letter = 'A';

printf("Character %c\n", first_letter); //display character

printf("ASCII value %d\n", first_letter); //display ASCII value

return 0;

}
```

Functions

- A function in C can perform a particular task, and supports the concept of modularity.
- A function is a self-contained block of statements that perform a coherent task of some kind.
- Making function is a way of isolating one block of code from other independent blocks of code.

Syntax of Function Definition:

return_data_type function_name (data_type variable1, data_type variable2, …)
{
    function_body
}

A *function definition* in C programming consists of a function header and a function body.

- **Return Type:** A function may return a value. The return_type is the data type of the value the function returns. Some functions may perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Syntax of Function Declaration:

```
return_type function_name( parameter list );
```

A function declaration tells the compiler about a function name and how to call the function.

Example-1: Multiplication of two numbers using function

```
#include <stdio.h>

int multiplication(int, int); //function declaration (prototype)
```

```
int multiplication(int x, int y)

{ x = x * y; return x; } //function definition

int main()

{

   int i, j, k;

   scanf("%d%d",&i, &j);

   k = multiplication(i, j); // function call

   printf("%d\n", k);

   return 0;

}
```

Example-2: Program to calculate factorial of given number

```
#include <stdio.h>

void factorial( int );    /* ANSI function prototype */

void factorial( int n )

{

    int  i, factorial_number = 1;

    for( i = 1; i <= n; ++i )

      factorial_number *= i;

    printf("The factorial of %d is %d\n", n, factorial_number );

}


main()

{

        int  number = 0;

        printf("Enter a number\n");
```

```
        scanf("%d", &number );

        factorial( number );

}
```

There are 4 types of functions based on return type:

1. Function with arguments but no return value
2. Function with arguments and return value
3. Function without arguments and no return value
4. Function without arguments but return value.

While calling a function in C program, there are two ways in which arguments can be passed to a function.

- **Call by Value:** If we pass values of variables to the function as parameters, such kind of function calling is known as **call by value**.
- **Call by Reference:** Variables are stored somewhere in memory. So, instead of passing the value of a variable, if we pass the location number / address of the variable to the function, then it would become ‘a call by reference’.

A function can call itself such a process is called recursion.

Functions can be of two types: (i) Library functions, and (ii) User-defined functions

Pointers

A pointer is a variable that stores memory address. Like all other variables, it also has a name, has to be declared and occupies some spaces in memory. It is called **pointer**because it points to a particular location.

- ‘&’ =Address of operator
- ‘*’ = Value at address operator or ‘indirection’ operator
- &i returns the address of the variable i.
- *(&i) return the value stored at a particular address printing the value of *(&i) is same as printing the value of i.

Pointers are useful due to following reasons:

- They enable us to access a variable that is defined outside a function.
- Pointers are more efficient in handling data tables and sometimes even arrays.
- Pointers tend to reduce the length and complexity of a program.
- They increase the execution speed.
- Use of pointers allows easy access to character strings.

NOTE:

- int *p; /* p is going to store address of integer value (size of memory is 2 bytes) */
- float *q; /* q is going to store address of floating value (size of memory is 2 bytes) */
- char *ch; /*ch is going to store address of character variable (size of memory is 2 bytes) */

Pointer Declaration: data-type *pointer-name;

NULL Pointers: Uninitilized pointers start out with random unknown values, just like any other variable type.

- Accidentally using a pointer containing a random address is one of the most common errors encountered when using pointers, and potentially one of the hardest to diagnose, since the errors encountered are generally not repeatable.

Example: Program to swap two integer numbers using pointers.

```
#include <stdio.h>

void swap(int *a,int *b);

int main()

{

   int a,b;

   a = 5;

   b = 10;

   printf("\nBefore swapping a= %d: b= %d", a, b);

   swap(&a, &b); //call function

   printf("\nAfter swapping a= %d: b= %d", a, b);

   return 0;

}


void swap(int *a, int *b)

{

   int x;

   x = *b;

   *b = *a;

   *a = x;

}
```

Pointers and Strings:

An array of characters is called a string. Strings in C are handled differently than most other languages. A pointer is used to keep track of a text string stored in memory. It will point to the first character of the string. By knowing the beginning address and the length of the string, the program can locate it.

- Example: char *a; a = “Hello World!”;

Constant pointer: A pointer is said to be constant pointer when the address its pointing to cannot be changed.

- Syntax: <type-of-pointer> *const <name-of-pointer>
- Example: int * const p;

C Pointer to Constant: This type of pointer cannot change the value at the address pointed by it.

- Syntax: const <type-of-pointer> *<name-of-pointer>;
- Example: const int * p;

Pointer to Pointer: It is a special pointer variable that can store the address of an other pointer variable. This means that its perfectly legal for a pointer to be pointing to another pointer.

1. Syntax: <type-of-pointer> ** <name-of-pointer>;
2. Example: int **p;

Array of Pointers: An array of pointers can be declared as follows.

- Declaration Syntax: <type> *<name>[<number-of-elements];
- Example: int *p[3];

Pointer to an Array: Pointer to an array can be declared as follows.

- Declaration Syntax: <type> (*<name>) [<number-of-elements];
- Example: int (*p)[3];

Function Pointers: Function pointer is same as pointer to a function.

- Syntax: <return type of function> (*<name of pointer>) (type of function arguments);
- Example: int (*f)(int, int); //declaration .

Combinations of * and ++

- *p++ accesses the thing pointed to by p and increments p
- (*p)++ accesses the thing pointed to by p and increments the thing pointed to by p
- *++p increments p first, and then accesses the thing pointed to by p
- ++*p increments the thing pointed to by p first, and then uses it in a larger expression.

Pointer Operations:

- **Assignment:** You can assign an address to a pointer. Typically, you do this by using an array name or by using the address operator (&).

- **Value finding (dereferencing):** The * operator gives the value stored in the pointed-to location.
- **Taking a pointer address:** Like all variables, pointer variables have an address and a value. The & operator tells you where the pointer itself is stored.
- **Adding an integer to a pointer:** You can use the + operator to add an integer to a pointer or a pointer to an integer. In either case, the integer is multiplied by the number of bytes in the pointed-to type, and the result is added to the original address.
- **Incrementing a pointer:** Incrementing a pointer to an array element makes it move to the next element of the array.
- **Subtracting an integer from a pointer:** You can use the – operator to subtract an integer from a pointer; the pointer has to be the first operand or a pointer to an integer. The integer is multiplied by the number of bytes in the pointed-to type, and the result is subtracted from the original address.
- Note that there are two forms of subtraction. You can subtract one pointer from another to get an integer, and you can subtract an integer from a pointer and get a pointer.
- **Decrementing a pointer:** You can also decrement a pointer. In this example, decrementing ptr2 makes it point to the second array element instead of the third. Note that you can use both the prefix and postfix forms of the increment and decrement operators.
- **Subtraction:** You can find the difference of two pointers. Normally, you do this for two pointers to elements that are in the same array to find out how far apart the elements are. The result is in the same units as the type size.
- **Comparisons:** You can use the relational operators to compare the values of two pointers, provided the pointers are of the same type.

Problems with Pointers:

- Assigning Value to Pointer Variable
- Assigning Value to Uninitialized Pointer
- Not de-referencing Pointer Variable (forgetting to use * before vaiable)
- Assigning Address of Un-initialized Variable
- Comparing Pointers that point to different objects
- Dangling pointers (Using free or de-allocating memory or out of scope)

Recursion

A function that calls itself directly or indirectly is called a recursive function. The recursive factorial function uses more memory than its non-recursive counter part. Recursive function requires stack support to save the recursive function calls.

- Recursion leads to compact
- It is simple
- It is easy to understand
- It is easy to prove correct

Example-1: Factorial Recursive Function

```
int factorial(int n)
{
        if (n == 0) return 1;
        else return n*factorial(n - 1);
}
```

Example-2: GCD Recursive Function

```
int gcd(int small, int large)
{
        if(small == 0) return 1;
        return gcd(1%s, s);
}
```

Example-3: Fibonacci Sequence Recursive Function

```
int Fibonacci(int x) {
  if (x == 0) return 1;  // Stopping conditions
  if (x == 1) return 1;
  return Fibonacci(x - 1) + Fibonacci(x - 2);
}
```

Example-4: Power Recursive Function (x^y)

```
double power(double x, unsigned y)
{
      if(y == 0)                    /* pow(x, 0) returns 1 */
            return(1.0);
      else
            return(power(x, y - 1) * x);
}
```

Example-5: Searching maximum value of an array with function call max(a, 0, n-1).

```
int mid, leftmax, rightmax;
int max (int a[], int low, int high)
{
if (low == high) return a[low];
else
{
mid = (low + high) / 2;
leftmax = max (a, low, mid);
rightmax = max (a, mid + 1, high);
if (leftmax > rightmax) return leftmax;
```

```
else return rightmax;
}
}
```

Example-6: Computing the sum of numbers from 1 to n

```
int sum (int n)
{
int s;
if (n == 0) return 0;
s = n + sum(n-1);
return s;
}
```

Arrays

- Array is a collection of similar elements having same data type, accessed using a common name.
- Array elements occupy contiguous memory locations.
- Array indices start at zero in C, and go to one less than the size of the array.

Declaration of an Array:

type variable[num_elements];

Example: int A[100];

- It creates an array A with 100 integer elements.
- The size of an array A can’t be changed.
- The number between the brackets must be a constant.

Initialization of an Array:

- int A[5]= {1,2,3,4,5}; /*Array can be initialized during declaration*/
- int A[5]={1,2,3}; /* Remaining elements are automatically initialized to zero*/
- int A[5]={1,[1]=2, 3,4,[4]=0};/* Array element can be initialized by specifying its index location*/

Problems with Arrays:

- There is no checking at run-time or compile-time to see whether reference is within array bounds.
- Size of array must be known at compile time.

Example-1: Read the 10 values into an array of size 10.

```
void main()
{
int A[10], i;
for (i=0; i<10; i++)
{
printf(“Enter the number %d”, i+1);
scanf(“%d”, &a[i]);
```

}

Example-2: Print the 10 values of an Array A.

void main()
{
int A[10], i;
for (i=0; i<10; i++)
{
printf("%d ", A[i]);
}
}

Pointers & Arrays: Let a[10] be an array with 10 elements.

- The name a of the array is a constant expression, whose value is the address of the 0th location.
- An array variable is actually just a pointer to the first element in the array.
- You can access array elements using array notation or pointers.
- a[0] is the same as *a
- a[1] is the same as *(a + 1)
- a[2] is the same as *(a + 2)
- a = a+0 = &a[0]
- a+1 = &a[1]
- a+i = &a[i]
- &(*(a+i)) = &a[i] = a+i
- *(&a[i]) = *(a+i) = a[i]
- Address of an element i of array a = a + i * sizeof(element)

C program to insert in an array

```
#include <stdio.h>

int main()
{
   int array[100], position, c, n, value;

   printf("Enter number of elements in array\n");
   scanf("%d", &n);

   printf("Enter %d elements\n", n);

   for (c = 0; c < n; c++)
      scanf("%d", &array[c]);

   printf("Enter the location where you wish to insert an element\n");
   scanf("%d", &position);

   printf("Enter the value to insert\n");
   scanf("%d", &value);

   for (c = n - 1; c >= position - 1; c--)
```

```
      array[c+1] = array[c];

   array[position-1] = value;

   printf("Resultant array is\n");

   for (c = 0; c <= n; c++)
      printf("%d\n", array[c]);

   return 0;
}

C Programming Delete From An array
#include <stdio.h>

int main()
{
   int array[100], position, c, n;

   printf("Enter number of elements in array\n");
   scanf("%d", &n);

   printf("Enter %d elements\n", n);

   for ( c = 0 ; c < n ; c++ )
      scanf("%d", &array[c]);

   printf("Enter the location where you wish to delete element\n");
   scanf("%d", &position);

   if ( position >= n+1 )
      printf("Deletion not possible.\n");
   else
   {
      for ( c = position - 1 ; c < n - 1 ; c++ )
         array[c] = array[c+1];

      printf("Resultant array is\n");

      for( c = 0 ; c < n - 1 ; c++ )
         printf("%d\n", array[c]);
   }

   return 0;
}
```

Multi-Dimensional Array

In C language, one can have arrays of any dimensions. Let us consider a 3 × 3 matrix

$$\begin{array}{ll|c|c|c|l} & & 0 & 1 & 2 & \text{Column number [j]} \\ \cline{3-5} \text{Row number [i]} & 0 & a_{0,0} & a_{0,1} & a_{0,2} & \\ \cline{3-5} & 1 & a_{1,0} & a_{1,1} & a_{1,2} & \\ \cline{3-5} & 2 & a_{2,0} & a_{2,1} & a_{2,2} & \\ \cline{3-5} \end{array}$$

3 × 3 matrix for multi-dimensional array

To access the particular element from the array, we have to use two subscripts; one for row number and other for column number. The notation is of the form a [i] [j], where i stands for row subscripts and j stands for column subscripts.
We can also define and initialize the array as follows

```
int values [3] [4] = {
                        {1, 2, 3, 4,}
                        {5, 6, 7, 8}
                        {9, 10, 11, 12}
                     };
        OR

 int values [3] [4]
           = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Note: Two Dimensional Array b[i][j]

- For Row Major Order: Size of b[i][j] = b + (Number of rows * i + j)*sizeof(element)
- For Column Major Order: Size of b[i][j] = b + (Number of Columns * j + i)*sizeof(element)
- *(*(b + i) + j) is equivalent to b[i][j]
- *(b + i) + j is equivalent to &b[i][j]
- *(b[i] + j) is equivalent to b[i][j]
- • b[i] + j is equivalent to &b[i][j]
- (*(b+i))[j] is equivalent to b[i][j]

Strings

In C language, strings are stored in an array of character (char) type along with the null terminating character "\0″ at the end.

Printf and scanf use "%s" format character for string. Printf print characters up to terminating zero. Scanf read characters until whitespace, store result in string, and terminate with zero.

Example:

char name[ ] = { 'G', 'A', 'T','E', 'T', 'O', 'P', '\O'};
OR
char name[ ] = "GATETOP";

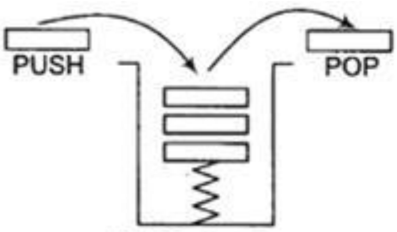
- '\0′ = Null character whose ASCII value is O.
- '0' = ASCII value is 48.
- In the above declaration '\0′ is not necessary. C inserts the null character automatically.

```
# include <stdio.h>
void main ( )
  {
        char name [ ] = "RAM";
        printf ("%S", name);
  }
```

- %s is used in printf ( ) as a format specification for printing out a string.
- All the following notations refer to the same element: name [i] , * (name + i), * (i + name), i [name]

Stacks

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the TOP of the stack. It is a LIFO (Last In First Out) kind of data structure.



A LIFO diagram

Operations on Stack

- **Push:** Adds an item onto the stack. PUSH (s, i); Adds the item i to the top of stack.
- **Pop:** Removes the most-recently-pushed item from the stack. POP (s); Removes the top element and returns it as a function value.
- size(): It returns the number of elements in the queue.
- isEmpty(): It returns true if queue is empty.

Implementation of Stack

A stack can be implemented in two ways: Using Array and Using Linked list.
But since array sized is defined at compile time, it can't grow dynamically. Therefore, an attempt to insert/push an element into stack (which is implemented through array) can cause a stack overflow situation, if it is already full.
Go, to avoid the above mentioned problem we need to use linked list to implement a stack, because linked list can grow dynamically and shrink at runtime.

1. Push and Pop Implementation Using Array:

```
void push( ) {
if(top==max)
printf("\nOverflow");
else {
int element;
printf("\nEnter Element:");
scanf("%d",&element);
printf("\nElement(%d) has been pushed at %d", element, top);
stack[top++]=element;
}
}
void pop( ) {
if(top==-1)
printf("\nUnderflow");
else
{
top–;
printf("\nELement has been popped out!");
}
}
```

2. Push and Pop Implementation Using Linked List:

```
struct node {
int data;
struct node *prev;
}*top=NULL, *temp=NULL;
void push( ) {
temp = (struct node*)malloc(sizeof(struct node*));
printf("\nEnter Data:");
scanf("%d",&temp->data);
temp->prev=NULL;
if(top==NULL) {
top=temp;
}
else {
temp->prev=top;
top=temp;
}
}
```

```
void pop( ) {
temp=top->prev;
top=temp;
printf("\nDeleted: %d",top->prev);
}
```

Applications of Stack

- Backtracking: This is a process when you need to access the most recent data element in a series of elements.
- Depth first Search can be implemented.
- The function call mechanism.
- Simulation of Recursive calls: The compiler uses one such data structure called stack for implementing normal as well as recursive function calls.
- Parsing: Syntax analysis of compiler uses stack in parsing the program.
- Web browsers store the addresses of recently visited sites on a stack.
- The undo-mechanism in an editor.
- Expression Evaluation: How a stack can be used for checking on syntax of an expression.
    - **Infix expression:** It is the one, where the binary operator comes between the operands.
      e. g., A + B * C.
    - **Postfix expression:** Here, the binary operator comes after the operands.
      e.g., ABC * +
    - **Prefix expression:** Here, the binary operator proceeds the operands.
      e.g.,+ A * BC
    - This prefix expression is equivalent to A + (B * C) infix expression. Prefix notation is also known as Polish notation. Postfix notation is also known as suffix or Reverse Polish notation.
- **Reversing a List:** First push all the elements of string in stack and then pop elements.
- **Expression conversion:** Infix to Postfix, Infix to Prefix, Postfix to Infix, and Prefix to Infix
- Implementation of **Towers of Hanoi**
- Computation of a cycle in the graph

Example-1: Implementation of Towers of Hanoi

Let A, B, and C be three stacks. Initially, B and C are empty, but A is not.
Job is to move the contents of A onto B without ever putting any object x on top of another object that was above x in the initial setup for A.

```
void TOH (int n, Stack A, Stack B, Stack C) {
if (n == 1) B.push (A.pop());
else {
TOH (n – 1, A, C, B); // n-1 go from A onto C
B.push (A.pop());
TOH (n – 1, C, B, A); // n-1 go from C onto B
}
}
```

Example-2: Evaluate the following postfix notation of expression: 15 3 2 + / 7 + 2 *

Ans. The stack operation for the postfix expression is:

Scanned elements	Operation	Stack
15	PUSH 15	15
3	PUSH 3	15, 3
2	PUSH 2	15, 3, 2
+	POP 2 and POP 3	
	Calculate: 3 + 2 = 5	
	PUSH 5	15, 5
/	POP 5 and POP 15	
	Calculate: 15 / 5 = 3	
	PUSH 3	3
7	PUSH 7	3, 7
+	POP 7 and POP 3	
	Calculate: 7 + 3 = 10	
	PUSH 10	10
2	PUSH 2	10, 2
*	POP 2 and POP 10	
	Calculate: 10 * 2 = 20	
	PUSH 20	20

Queues

It is a non-primitive, linear data structure in which elements are added/inserted at one end (called the REAR) and elements are removed/deleted from the other end (called the FRONT). A queue is logically a FIFO (First in First Out) type of list.

Operations on Queue

- **Enqueue:** Adds an item onto the end of the queue ENQUEUE(Q, i); Adds the item i onto the end of queue.
- **Dequeue:** Removes the item from the front of the queue. DEQUEUE (Q); Removes the first element and returns it as a function value.

Queue Implementation: Queue can be implemented in two ways.

- Static implementation (using arrays)
- Dynamic implementation (using linked lists)

Queue Implementation Using Arrays

```
void enqueue()
{
int element;
if(rear==max)
{
printf("\nOverflow!!");
}
else
{
printf("\nEnter Element:");
scanf("%d",&element);
queue[rear++]=element;
printf("\n %d Enqueued at %d",element,rear);
}
}
void dequeue()
{
if(rear==front)
{
printf("\nUnderflow!!");
}
else
{
front++;
printf("\nElement is Dequeued from %d",front);
}
}
```

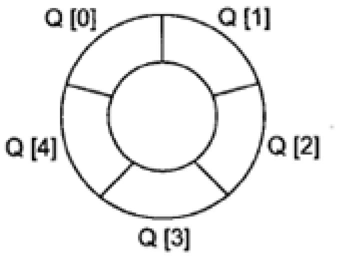
Queue Implementation Using Linked Lists

```
typedef struct qnode
{
int data;
struct qnode *link;
}node;
node *front=NULL;
node *rear=NULL;void enqueue()
{
int item;
node *temp;
printf("Enter the item\n");
scanf("%d",&item);
temp=(node*)malloc(sizeof(node));
```

```
temp->data=item;
temp->link=NULL;
if(rear==NULL)
{
front=temp;
rear=temp;
}
else
{
rear->link=temp;
rear=temp;
}
}void dequeue()
{
int item;
if(front==NULL)
printf(“Queue is empty\n”);
else
{
item=front->data;
printf(“The element deleted = %d\n”,item);
}
if(front==rear)
{
front=NULL;
rear=NULL;
}
else
front=front->link;
}
```

Circular Queue

In a circular queue, the first element comes just after the last element or a circular queue is one in which the insertion of a new element is done at the very first location of the queue, if the last location of queue is full and the first location is empty.



Circular queue

Note: A circular queue overcomes the problem of unutilised space in linear queues implemented as arrays.

We can make following assumptions for circular queue.

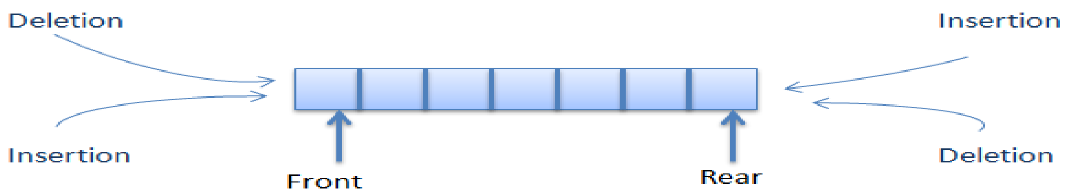
- If : (Rear+1) % n == Front, then queue is Full

- If Front = Rear, the queue will be empty.
- Each time a new element is inserted into the queue, the Rear is incremented by 1.
  Rear = (Rear + 1)) % n
- Each time, an element is deleted from the queue, the value of Front is incremented by one.
  Front = (Front + 1) % n

```
 #define SIZE 5            /* Size of Circular Queue */

int CQ[SIZE], f = -1, r = -1; /* Global declarations */

CQinsert(int elem) { /* Function for Insert operation */
 if (CQfull())
  printf("\n\n Overflow!!!!\n\n");
 else {
  if (f == -1)
   f = 0;
  r = (r + 1) % SIZE;
  CQ[r] = elem;
 }
}

int CQdelete() { /* Function for Delete operation */
 int elem;
 if (CQempty()) {
  printf("\n\nUnderflow!!!!\n\n");
  return (-1);
 } else {
  elem = CQ[f];
  if (f == r) {
   f = -1;
   r = -1;
  } /* Q has only one element ? */
  else
   f = (f + 1) % SIZE;
  return (elem);
 }
}

int CQfull() { /* Function to Check Circular Queue Full */
 if ((f == r + 1) || (f == 0 && r == SIZE - 1))
  return 1;
 return 0;
}

int CQempty() { /* Function to Check Circular Queue Empty */
 if (f == -1)
  return 1;
 return 0;
}

display() { /* Function to display status of Circular Queue */
 int i;
 if (CQempty())
  printf(" \n Empty Queue\n");
 else {
  printf("Front[%d]->", f);
  for (i = f; i != r; i = (i + 1) % SIZE)
   printf("%d ", CQ[i]);
  printf("%d ", CQ[i]);
```

```
  printf("<-[%d]Rear", r);
 }
}

main() { /* Main Program */
 int opn, elem;
 do {
  clrscr();
  printf("\n ### Circular Queue Operations ### \n\n");
  printf("\n Press 1-Insert, 2-Delete,3-Display,4-Exit\n");
  printf("\n Your option ? ");
  scanf("%d", &opn);
  switch (opn) {
  case 1:
   printf("\n\nRead the element to be Inserted ?");
   scanf("%d", &elem);
   CQinsert(elem);
   break;
  case 2:
   elem = CQdelete();
   if (elem != -1)
    printf("\n\nDeleted Element is %d \n", elem);
   break;
  case 3:
   printf("\n\nStatus of Circular Queue\n\n");
   display();
   break;
  case 4:
   printf("\n\n Terminating \n\n");
   break;
  default:
   printf("\n\nInvalid Option !!! Try Again !! \n\n");
   break;
  }
  printf("\n\n\n\n  Press a Key to Continue . . . ");
  getch();
 } while (opn != 4);
}
```

Double Ended Queue (DEQUE): **It is a list of elements in which insertion and deletion operations are performed from both the ends. That is why it is called double-ended queue or DEQUE.**



Algorithm for Insertion at rear end

Step -1: [Check for overflow]

```
    if(rear==MAX)
        Print("Queue is Overflow");
  return;
Step-2: [Insert element]
    else
        rear=rear+1;
        q[rear]=no;
        [Set rear and front pointer]
        if rear=0
            rear=1;
        if front=0
            front=1;
```

Step-3: return

Algorithm for Insertion at font end

Step-1 : [Check for the front position]

```
    if(front<=1)
        Print ("Cannot add item at front end");
    return;
```

Step-2 : [Insert at front]

```
    else
        front=front-1;
        q[front]=no;
```

Step-3 : Return

Priority Queues: This type of queue enables us to retrieve data items on the basis of priority associated with them. Below are the two basic priority queue choices**.**

Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

- **remove / dequeue** – remove an item from the front of the queue.

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one

Sorted Array or List

It is very efficient to find and delete the smallest element. Maintaining sorted ness make the insertion of new elements slow.

Applications of Queue

- Breadth first Search can be implemented.
- CPU Scheduling
- Handling of interrupts in real-time systems
- Routing Algorithms
- Computation of shortest paths
- Computation a cycle in the graph

Example-1: Reversing the Queue Q using the Stack S

```
void reverse( Queue ∗Q) {
Stack S ; //Assume Empty Stack S is created
whil e ( ! isEmpty (Q) ) {
S.push(&S , deQueue (Q ) ) ;
}
whil e ( ! isEmpty(&S ) ) {
enQueue (Q, pop(&S ) ) ;
}
}
```

Example-2: Find the effect of the following code with Circular Queue Q having locations from 0 to 6.

```
for (int k = 1; k <= 7; k++)
Q.enqueue(k);
for (int k = 1; k <= 4; k++) {
Q.enqueue(Q.dequeue());
```

Q.dequeue();
}
Answer: After the above code execution on empty queue will result the following elements.

- 3 is stored at location 1,
- 5 is stored at location 2, and
- 7 is stored at location 3.

Implementation of Queue Using Two Stacks
Method 1: Let S1 and S2 be the two stacks to be used in the implementation of queue Q.

```
Enqueue(int a){
S1.push(a);
}
int Dequeue( ){
if (S1 is empty) return(error);
while(S1 is not empty){
S2.push(S1.pop());
}
r = S2.pop();
while(S2 is not empty){
S1.push(S2.pop());
}
return(r);
}
```

Method2: Let S1 and S2 be the two stacks to be used in the implementation of queue Q.

```
Enqueue(int a){
S1.push(a);
}
int dequeue( ){
if (S1 is empty & S2 is empty) return(error);
if (S2 is empty){
while(S1 is not empty){
S2.push(S1.pop());
}
}
return(S2.pop());
}
```

Implementation of Stack Using two Queues
Method 1: Let Q1 and Q2 be two queues.

- push:
  - Enqueue in queue1
- pop:
  - while size of queue1 is bigger than 1, pipe dequeued items from queue1 into queue2
  - Dequeue and return the last item of queue1, then switch the names of queue1 and queue2

Push is constant time but Pop operation is O(n) time

```
void push(int data){
Enqueue(Q1, data)
}
int pop(){
int returnValue =-1; // indicate Stack Empty.
while(!isEmpty(Q1))
{
returnValue = Dequeue(Q1);
// If it was last element of queue1. return it.
if(isEmpty(Q1))
break;
else
Enqueue(Q1, returnValue);
}
// swap the names of queue1 and queue2.
// If swapping is not possible then we will have to move all the elements from queue2 to queue1
// or have another flag to indicate the active queue.
Node * temp = Q1;
Q1 = Q2;
Q2 = temp;
return returnValue;
}
```

Method 2:

- push:
    - Enqueue in queue2
    - Enqueue all items of queue1 in queue2, then switch the names of queue1 and queue2
- pop:
    - Deqeue from queue1

Pop is constant time but Push operation is O(n) time

```
void push(int data){
Enqueue(Q2, data);
while(!isEmpty(Q1)){
Enqueue(Q2, Dequeue(Q1));
}
// swap the names of queue1 and queue2.
// If swapping is not possible then we will have to move all the elements from queue2 to queue1
// or have another flag to indicate the active queue.
Node * temp = Q1;
Q1 = Q2;
Q2 = temp;
```

}
// Put proper check to see if no element in Queues
int pop(){
return Dequeue(Q1);

Linked Lists

Linked list is a special data structure in which data elements are linked to one another. Here, each element is called a node which has two parts

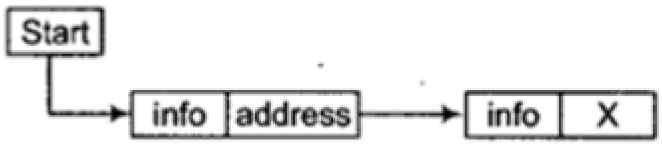
- Info part which stores the information.
- Address or pointer part which holds the address of next element of same type.Linked list is also known as self-referential structure.

Each element (node) of a list is comprising of two items: the data and a reference to the next node.

- The last node has a reference to NULL.
- The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate
- node, but the reference to the first node.
- If the list is empty then the head is a null reference.

```
struct linked_list_node
    {
      int info;
      struct lined_list_node *next;
    };
```

Syntax of declaring a node which contains two fields in it one is for storing information and another is for storing address of other node, so that one can traverse the list.



Linked list process flow diagram

Advantages of Linked List:

- Linked lists are dynamic data structure as they can grow and shrink during the execution time.
- Efficient memory utilisation because here memory is not pre-allocated.
- Insertions and deletions can be done very easily at the desired position.

Disadvantages of Linked List:

- More memory is required, if the number of fields are, more.
- Access to an arbitrary data item is time consuming.

Operations on Linked Lists: The following operations involve in linked list are as given below

- **Creation:** Used to create a lined list.

- **Insertion:** Used to insert a new node in linked list at the specified position. A new node may be inserted
    - At the beginning of a linked list
    - At the end of a linked list
    - At the specified position in a linked list
    - In case of empty list, a new node is inserted as a first node.
- **Deletion:** This operation is basically used to delete as item (a node). A node may be deleted from the
    - Beginning of a linked list.
    - End of a linked list.
    - Specified position in the list.
- **Traversing:** It is a process of going through (accessing) all the nodes of a linked list from one end to the other end.

Types of Linked Lists

- **Singly Linked List:** In this type of linked list, each node has only one address field which points to the next node. So, the main disadvantage of this type of list is that we can't access the predecessor of node from the current node.
- **Doubly Linked List:** Each node of linked list is having two address fields (or links) which help in accessing both the successor node (next node) and predecessor node (previous node).
- **Circular Linked List:** It has address of first node in the link (or address) field of last node.
- **Circular Doubly Linked List:** It has both the previous and next pointer in circular manner.

```
 /* Program of single linked list*/
# include <stdio.h>
# include <malloc.h>

struct node
{
	int info;
	struct node *link;
}*start;

main()
{
	int choice,n,m,position,i;
	start=NULL;
	while(1)
	{
		printf("1.Create List\n");
		printf("2.Add at begining\n");
		printf("3.Add after \n");
		printf("4.Delete\n");
		printf("5.Display\n");
```

```
printf("6.Count\n");
printf("7.Reverse\n");
printf("8.Search\n");
printf("9.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
 case 1:
        printf("How many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter the element : ");
                scanf("%d",&m);
                create_list(m);
        }
        break;
 case 2:
        printf("Enter the element : ");
        scanf("%d",&m);
        addatbeg(m);
        break;
 case 3:
        printf("Enter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&position);
        addafter(m,position);
        break;
 case 4:
        if(start==NULL)
        {
                printf("List is empty\n");
                continue;
        }
        printf("Enter the element for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
 case 5:
        display();
        break;
 case 6:
        count();
        break;
 case 7:
        rev();
```

```
                        break;
                case 8:
                        printf("Enter the element to be searched : ");
                        scanf("%d",&m);
                        search(m);
                        break;
                case 9:
                        exit();
                default:
                        printf("Wrong choice\n");
                }/*End of switch */
        }/*End of while */
}/*End of main()*/

create_list(int data)
{
        struct node *q,*tmp;
        tmp= malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=NULL;

        if(start==NULL) /*If list is empty */
                start=tmp;
        else
        {       /*Element inserted at the end */
                q=start;
                while(q->link!=NULL)
                        q=q->link;
                q->link=tmp;
        }
}/*End of create_list()*/

addatbeg(int data)
{
        struct node *tmp;
        tmp=malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=start;
        start=tmp;
}/*End of addatbeg()*/

addafter(int data,int pos)
{
        struct node *tmp,*q;
        int i;
        q=start;
        for(i=0;i<pos-1;i++)
        {
```

```
			q=q->link;
			if(q==NULL)
			{
				printf("There are less than %d elements",pos);
				return;
			}
		}/*End of for*/

		tmp=malloc(sizeof(struct node) );
		tmp->link=q->link;
		tmp->info=data;
		q->link=tmp;
}/*End of addafter()*/

del(int data)
{
		struct node *tmp,*q;
		if(start->info == data)
		{
			tmp=start;
			start=start->link;  /*First element deleted*/
			free(tmp);
			return;
		}
		q=start;
		while(q->link->link != NULL)
		{
			if(q->link->info==data)     /*Element deleted in between*/
			{
				tmp=q->link;
				q->link=tmp->link;
				free(tmp);
				return;
			}
			q=q->link;
		}/*End of while */
		if(q->link->info==data)    /*Last element deleted*/
		{
			tmp=q->link;
			free(tmp);
			q->link=NULL;
			return;
		}
		printf("Element %d not found\n",data);
}/*End of del()*/

display()
{
```

```
	struct node *q;
	if(start == NULL)
	{
		printf("List is empty\n");
		return;
	}
	q=start;
	printf("List is :\n");
	while(q!=NULL)
	{
		printf("%d ", q->info);
		q=q->link;
	}
	printf("\n");
}/*End of display() */

count()
{
	struct node *q=start;
	int cnt=0;
	while(q!=NULL)
	{
		q=q->link;
		cnt++;
	}
	printf("Number of elements are %d\n",cnt);
}/*End of count() */

rev()
{
	struct node *p1,*p2,*p3;
	if(start->link==NULL)    /*only one element*/
		return;
	p1=start;
	p2=p1->link;
	p3=p2->link;
	p1->link=NULL;
	p2->link=p1;
	while(p3!=NULL)
	{
		p1=p2;
		p2=p3;
		p3=p3->link;
		p2->link=p1;
	}
	start=p2;
}/*End of rev()*/
```

```
search(int data)
{
	struct node *ptr = start;
	int pos = 1;
	while(ptr!=NULL)
	{
		if(ptr->info==data)
		{
			printf("Item %d found at position %d\n",data,pos);
			return;
		}
		ptr = ptr->link;
		pos++;
	}
	if(ptr == NULL)
		printf("Item %d not found in list\n",data);
}/*End of search()*/
```

Example-1: Reverse of a Singly Linked List with iterations

```
void reverse_list() {
node *p, *q, *r;
if(head == (mynode *)0) { return; }
p = head;
q = p->next;
p->next = (mynode *)0;
while (q != (mynode *)0) {
r = q->next;
q->next = p;
p = q;
q = r;
}
head = p;
}
```

Example2: Reverse of a Singly Linked List with Recursion

```
node* reverse_list(mynode *root) {
if(root->next!=(mynode *)0) {
reverse_list(root->next);
root->next->next=root;
return(root);
}
else {
head=root;
}
}
```

Example-3: Reverse of a Doubly Linked List

```
void reverse( ) {
node *cur, *temp, *nextnode;
```

```
if(head==tail) return;
if(head==NULL || tail==NULL) return;
for(cur=head; cur!=NULL; ) {
temp=cur->next;
nextnode=cur->next;
cur->next=cur->prev;
cur->prev=temp;
cur=nextnode;
}
temp=head;
head=tail;
tail=temp;
}
```

Example-4: Finding the middle of a Linked List

```
struct node *middle(struct node *head) {
p=head;
q=head;
if(q->next->next!=NULL) {
p=p->next;
q=q->next->next;
}
return p;
}
```

Time complexity for the following operations on Singly Linked Lists of n nodes:

- Add a new node to the beginning of list: O(1)
- Add a new node to the end: O(n)
- Add a new node after k'th node: O(n)
- Search a node with a given data: O(n)
- Add a new node after a node with a given data: O(n)
- Add a new node before a node with a given data: O(n)
- Traverse all nodes: O(n)
- Modify the data of all nodes in a linked list: O(n)

Time complexity for the following operations on Doubly Linked Lists of n nodes:

- Add a new node to the beginning of list: O(1)
- Add a new node to the end: O(n)
- Add a new node after k'th node: O(n)
- Search a node with a given data: O(n)
- Add a new node after a node with a given data: O(n)
- Add a new node before a node with a given data: O(n)
- Traverse all nodes: O(n)
- Modify the data of all nodes in a linked list: O(n)

Time complexity for the following operations on Circular Singly Linked Lists of n nodes:

- Add a new node to the beginning of list: O(n)

- Add a new node to the end: O(n)
- Add a new node after k'th node: O(n)
- Search a node with a given data: O(n)
- Add a new node after a node with a given data: O(n)
- Add a new node before a node with a given data: O(n)
- Traverse all nodes: O(n)
- Delete a node from the beginning: O(n)
- Modify the data of all nodes in a linked list: O(n)

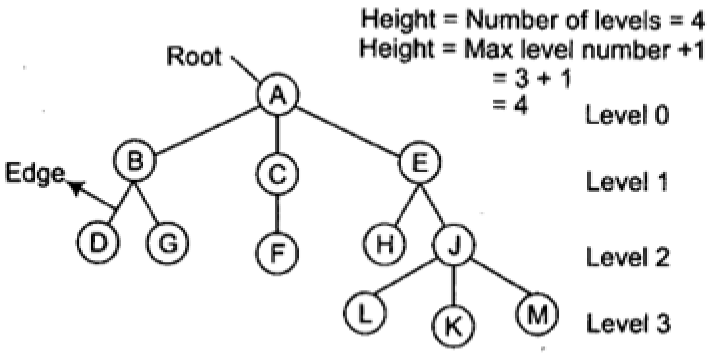
Time complexity for the following operations on Circular Doubly Linked Lists of n nodes:

- Add a new node to the beginning of list: O(1)
- Add a new node to the end: O(1)
- Add a new node after k'th node: O(n)
- Search a node with a given data: O(n)
- Add a new node after a node with a given data: O(n)
- Add a new node before a node with a given data: O(n)
- Traverse all nodes: O(n)
- Modify the data of all nodes in a linked list: O(n)

Trees

Tree is a non linear and hierarchical Data Structure.

Trees are used to represent data containing a hierarchical relationship between elements e. g., records, family trees and table contents. A tree is the data structure that is based on hierarchical tree structure with set of nodes.



A tree diagram

- **Node:** Each data item in a tree.
- **Root:** First or top data item in hierarchical arrangement.
- **Degree of a Node:** Number of subtrees of a given node.
  - Example: Degree of A = 3, Degree of E = 2
- **Degree of a Tree:** Maximum degree of a node in a tree.
  - Example: Degree of above tree = 3

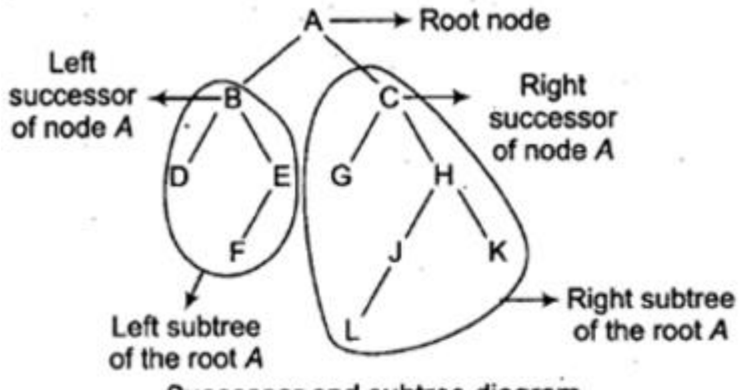
- **Depth or Height:** Maximum level number of a node + 1(i.e., level number of farthest leaf node of a tree + 1).
    - Example: Depth of above tree = 3 + 1= 4
- **Non-terminal Node:** Any node except root node whose degree is not zero.
- **Forest:** Set of disjoint trees.
- **Siblings:** D and G are siblings of parent Node B.
- **Path:** Sequence of consecutive edges from the source node to the destination node.
- **Internal nodes:** All nodes those have children nodes are called as internal nodes.
- **Leaf nodes:** Those nodes, which have no child, are called leaf nodes.
- The depth of a node is the number of edges from the root to the node.
- The height of a tree is the height of the root.

Trees can be used

- to represent Heaps (Priority Queues)
- to represent B-Trees (fast access to database)
- for storing hierarchies in organizations
- for file system

Binary Tree: A binary tree is a tree like structure that is rooted and in which each node has at most two children and each child of a node is designated as its left or right child. In this kind of tree, the maximum degree of any node is at most 2.
A binary tree T is defined as a finite set of elements such that

- T is empty (called NULL tree or empty tree).
- T contains a distinguished Node R called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.



Successor and subtree diagram

Any node N in a binary tree T has either 0, 1 or 2 successors. Level l of a binary tree T can have at most 2^l nodes.

- Number of nodes on each level i of binary tree is at most 2^i
- The number n of nodes in a binary tree of height h is atleast n = h + 1 and atmost n = $2^{h+1} - 1$, where h is the depth of the tree.
- Depth d of a binary tree with n nodes >= floor(lg n)
    - d = floor(lg N) ; lower bound, when a tree is a full binary tree
    - d = n – 1 ; upper bound, when a tree is a degenerate tree

Creation of a binary tree

```
void insert(node ** tree, int val) {
 node *temp = NULL;
 if(!(*tree)) {
   temp = (node *)malloc(sizeof(node));
   temp->left = temp->right = NULL;
   temp->data = val;
   *tree = temp;
   return;
 }

 if(val < (*tree)->data) {
      insert(&(*tree)->left, val);
   }
 else if(val > (*tree)->data) {
     insert(&(*tree)->right, val);
   }
 }
```

Search an element into binary tree

```
node* search(node ** tree, int val) {
 if(!(*tree)) {
   return NULL;
 }
 if(val == (*tree)->data) {
   return *tree;
```

```
 }
else if(val < (*tree)->data) {
   search(&((*tree)->left), val);
 }
else if(val > (*tree)->data){
   search(&((*tree)->right), val);
 }
}
```

Delete an element from binary tree

```
void deltree(node * tree) {
 if (tree) {
   deltree(tree->left);
   deltree(tree->right);
   free(tree);
 }
}
```

Extended Binary Trees: 2. Trees or Strictly Binary Trees
If every non-terminal node in a binary tree consist of non-empty left subtree and right subtree. In other words, if any node of a binary tree has either 0 or 2 child nodes, then such tree is known as **strictly binary tree or extended binary** tree or 2- tree.

Complete Binary Tree: A complete binary tree is a tree in which every level, except possibly the last, is completely filled.
A Complete binary tree is one which have the following properties

- Which can have 0, 1 or 2 children.
- In which first, we need to fill left node, then right node in a level.
- In which, we can start putting data item in next level only when the previous level is completely filled.
- A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes.

Tree Traversal: Three types of tree traversal are given below

- **Preorder**

- Process the root R.
- Traverse the left subtree of R in preorder.
- Traverse the right subtree of R in preorder.

```
/* Recursive function to print the elements of a binary tree with preorder traversal*/

void preorder(struct btreenode *node)

{

  if (node != NULL)

  {

    printf("%d", node->data);

    preorder(node->left);

    preorder(node->right);

  }

}
```

- **Inorder**
  - Traverse the left subtree of R in inorder.
  - Process the root R.
  - Traverse the right subtree of R in inorder.

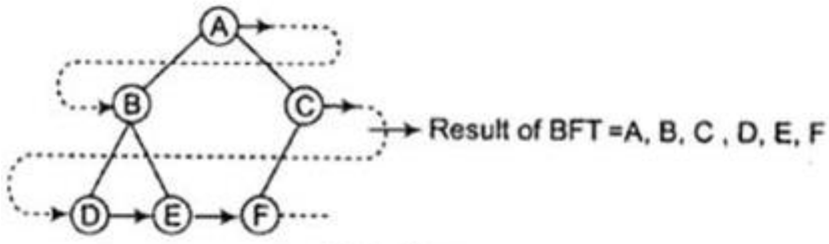
```
/* Recursive function to print the elements of a binary tree with inorder traversal*/

void inorder(struct btreenode *node)

{

  if (node != NULL)

  {

    inorder(node->left);

    printf("%d", node->data);

    inorder(node->right);

  }

}
```

- **Postorder**
    - Traverse the left subtree of R in postorder.
    - Traverse the right subtree of R in postorder.
    - Process the root R.

```
/* Recursive function to print the elements of a binary tree with postorder traversal*/

void postorder(struct btreenode *node)

{

  if (node != NULL)

  {

    postorder(node->left);

    postorder(node->right);

    printf("%d", node->data);

  }

}
```

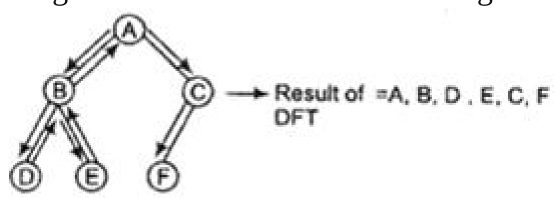
Breadth First Traversal (BFT): The breadth first traversal of a tree visits the nodes in the order of their depth in the tree.

BFT first visits all the nodes at depth zero (i.e., root), then all the nodes at depth 1 and so on. At each depth, the nodes are visited from left to right.



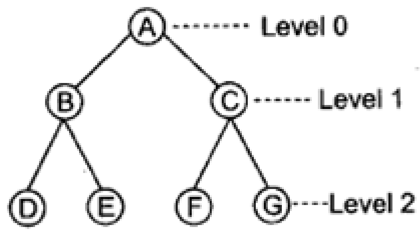
BFT diagram

Depth First Traversal (DFT): In DFT, one starts from root and explores as far as possible along each branch before backtracking.



A DFT diagram

Perfect Binary Tree or Full Binary Tree: A binary tree in which all leaves are at the same level or at the same depth and in which every parent has 2 children.



Full binary tree

Here, all leaves (D, E, F, G) are at depth 3 or level 2 and every parent is having exactly 2 children.

- Let a binary tree contain MAX, the maximum number of nodes possible for its height h. Then h= log(MAX + 1) –1.
- The height of the Binary Search Tree equals the number of links of the path from the root node to the deepest node.
- Number of internal/leaf nodes in a full binary tree of height *h*
  - 2^h leaves
  - 2^h -1 internal nodes

Expression Tree

An expression tree is a binary tree which represents a binary arithmetic expression. All internal nodes in the expression tree are **operators**, and leaf nodes are the operands. Expression tree will help in precedence relation of operators. (2+3)*4 and 2+(3*4) expressions will have different expression trees.

Example-1: Recursive function for size (number of nodes) of a binary tree

```
int size(struct btreenode *node)

{

 if (node == NULL)

   return 0;

 else

   return (1 + size(node->left) + size(node->right));

}
```

Example-2: Recursive function for Height of a tree

(Hieght is the length of path to the deepest node from the root node of tree)

```
int height(struct btreenode *node)

{
```

```
if (node == NULL) return 0;

else return (1 + Max(height(node->left), height(node->right)));

}
```

Example-3: Print the elements of binary tree using level order traversal

```
void levelorder(struct node* root)

{

 int rear, front;

 struct node **queue = createqueue(&front, &rear);

 struct node *tempnode = root;

 while (temp_node)

 {

 printf("%d ", tempnode->data);

 if (tempnode->left)

 enqueue(queue, &rear, tempnode->left);

 if (tempnode->right)

 enqueue(queue, &rear, tempnode->right);

 tempnode = dequeue(queue, &front);

 }

}

struct node** createqueue(int *front, int *rear)

{

 struct node **queue = (struct node **) malloc(sizeof(struct node*)*n);

 *front = *rear = 0;

 return queue;

}
```

Binary Search Trees

A binary tree T, is called binary search tree (or binary sorted tree), if each node N of T has the following property. The value at N is greater than every value in the left subtree of N and is less than or equal to every value in the right subtree of N. A BST holds the following properties:

- Each node can have up to two child nodes.
- The left subtree of a node contains only nodes with keys less than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.
- A unique path exists from the root to every other node.

```
/*Insertion ,Deletion and Traversal in Binary Search Tree*/
# include <stdio.h>
# include <malloc.h>

struct node
{
	int info;
	struct node *lchild;
	struct node *rchild;
}*root;

main()
{
	int choice,num;
	root=NULL;
	while(1)
	{
		printf("\n");
		printf("1.Insert\n");
		printf("2.Delete\n");
		printf("3.Inorder Traversal\n");
		printf("4.Preorder Traversal\n");
		printf("5.Postorder Traversal\n");
		printf("6.Display\n");
		printf("7.Quit\n");
```

```
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        printf("Enter the number to be inserted : ");
                        scanf("%d",&num);
                        insert(num);
                        break;
                 case 2:
                        printf("Enter the number to be deleted : ");
                        scanf("%d",&num);
                        del(num);
                        break;
                 case 3:
                        inorder(root);
                        break;
                 case 4:
                        preorder(root);
                        break;
                 case 5:
                        postorder(root);
                        break;
                 case 6:
                        display(root,1);
                        break;
                 case 7:
                        exit();
                 default:
                        printf("Wrong choice\n");
                }/*End of switch */
        }/*End of while */
}/*End of main()*/

find(int item,struct node **par,struct node **loc)
{
        struct node *ptr,*ptrsave;
```

```
	if(root==NULL)  /*tree empty*/
	{
		*loc=NULL;
		*par=NULL;
		return;
	}
	if(item==root->info) /*item is at root*/
	{
		*loc=root;
		*par=NULL;
		return;
	}
	/*Initialize ptr and ptrsave*/
	if(item<root->info)
		ptr=root->lchild;
	else
		ptr=root->rchild;
	ptrsave=root;

	while(ptr!=NULL)
	{
		if(item==ptr->info)
		{	*loc=ptr;
			*par=ptrsave;
			return;
		}
		ptrsave=ptr;
		if(item<ptr->info)
			ptr=ptr->lchild;
		else
			ptr=ptr->rchild;
	 }/*End of while */
	 *loc=NULL;   /*item not found*/
	 *par=ptrsave;
}/*End of find()*/

insert(int item)
{	struct node *tmp,*parent,*location;
	find(item,&parent,&location);
```

```
	if(location!=NULL)
	{
		printf("Item already present");
		return;
	}

	tmp=(struct node *)malloc(sizeof(struct node));
	tmp->info=item;
	tmp->lchild=NULL;
	tmp->rchild=NULL;

	if(parent==NULL)
		root=tmp;
	else
		if(item<parent->info)
			parent->lchild=tmp;
		else
			parent->rchild=tmp;
}/*End of insert()*/

del(int item)
{
	struct node *parent,*location;
	if(root==NULL)
	{
		printf("Tree empty");
		return;
	}

	find(item,&parent,&location);
	if(location==NULL)
	{
		printf("Item not present in tree");
		return;
	}

	if(location->lchild==NULL && location->rchild==NULL)
		case_a(parent,location);
	if(location->lchild!=NULL && location->rchild==NULL)
```

```
			case_b(parent,location);
		if(location->lchild==NULL && location->rchild!=NULL)
			case_b(parent,location);
		if(location->lchild!=NULL && location->rchild!=NULL)
			case_c(parent,location);
		free(location);
}/*End of del()*/

case_a(struct node *par,struct node *loc )
{
		if(par==NULL) /*item to be deleted is root node*/
			root=NULL;
		else
			if(loc==par->lchild)
				par->lchild=NULL;
			else
				par->rchild=NULL;
}/*End of case_a()*/

case_b(struct node *par,struct node *loc)
{
		struct node *child;

		/*Initialize child*/
		if(loc->lchild!=NULL) /*item to be deleted has lchild */
			child=loc->lchild;
		else		/*item to be deleted has rchild */
			child=loc->rchild;

		if(par==NULL )   /*Item to be deleted is root node*/
			root=child;
		else
			if( loc==par->lchild)   /*item is lchild of its parent*/
				par->lchild=child;
			else		/*item is rchild of its parent*/
				par->rchild=child;
}/*End of case_b()*/

case_c(struct node *par,struct node *loc)
```

```
{
	struct node *ptr,*ptrsave,*suc,*parsuc;

	/*Find inorder successor and its parent*/
	ptrsave=loc;
	ptr=loc->rchild;
	while(ptr->lchild!=NULL)
	{
		ptrsave=ptr;
		ptr=ptr->lchild;
	}
	suc=ptr;
	parsuc=ptrsave;

	if(suc->lchild==NULL && suc->rchild==NULL)
		case_a(parsuc,suc);
	else
		case_b(parsuc,suc);

	if(par==NULL) /*if item to be deleted is root node */
		root=suc;
	else
		if(loc==par->lchild)
			par->lchild=suc;
		else
			par->rchild=suc;

	suc->lchild=loc->lchild;
	suc->rchild=loc->rchild;
}/*End of case_c()*/

preorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return;
	}
	if(ptr!=NULL)
```

```
	{
		printf("%d  ",ptr->info);
		preorder(ptr->lchild);
		preorder(ptr->rchild);
	}
}/*End of preorder()*/

inorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return;
	}
	if(ptr!=NULL)
	{
		inorder(ptr->lchild);
		printf("%d  ",ptr->info);
		inorder(ptr->rchild);
	}
}/*End of inorder()*/

postorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return;
	}
	if(ptr!=NULL)
	{
		postorder(ptr->lchild);
		postorder(ptr->rchild);
		printf("%d  ",ptr->info);
	}
}/*End of postorder()*/

display(struct node *ptr,int level)
{
```

```
        int i;
        if ( ptr!=NULL )
        {
                display(ptr->rchild, level+1);
                printf("\n");
                for (i = 0; i < level; i++)
                        printf("    ");
                printf("%d", ptr->info);
                display(ptr->lchild, level+1);
        }/*End of if*/
}/*End of display()*/
```

Traversals of Binary Search Tree

Inorder Tree Walk: During this type of walk, we visit the root of a subtree between the left subtree visit and right subtree visit.

Inorder (*x*):

If $x \neq$ NIL {

  Inorder (left[*x*]);

  print key[*x*];

  Inorder (right[*x*]);

}

Preorder Tree Walk: In which we visit the root node before the nodes in either subtree.

Preorder (*x*):

If x ≠ NIL {

  print key[*x*];

  Preorder (left[*x*]);

  Preorder (right[*x*]);

}

Postorder Tree Walk: In which we visit the root node after the nodes in its subtrees.

Postorder(x):
If x ≠NIL {
Postorder (left[*x*]);
Postorder (right[*x*]);
print key [*x*];
}

Search an element in BST: The most basic operator is search, which can be a recursive or an iterative function. A search can start from any node, If the node is NULL (i.e. the tree is empty), then return NULL which means the key does not exist in the tree. Otherwise, if the key equals that of the node, the search is successful and we return the node. If the key is less than that of the node, we search its left subtree. Similarly, if the key is greater than that of the node, we search its right subtree. This process is repeated until the key is found or the remaining subtree is null. To search the key in the BFS, just call the method from the root node.

Insertion of an element: Insertion begins as a search would begin; We examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than the root. If the key exists, we can either replace the value by the new value or just return without doing anything.

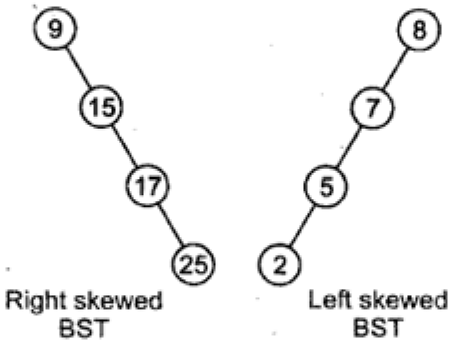
Deletion of an element: The deletion is a little complex. Basically, to delete a node by a given key, we need to find the node with the key, and remove it from the tree. There are three possible cases to consider:

- Deleting a leaf: we can simply remove it from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: find its in-order successor (left-most node in its right sub-tree), let's say R. Then copy R's key and value to the node, and remove R from its right sub-tree.

Key Points of BST

- It takes θ (n) time to walk (inorder, preorder and pastorder) a tree of n nodes.
- On a binary search tree of height h, Search, Minimum, Maximum, Successor, Predecessor, Insert, and Delete can be made to run in O(h) time.
- The height of the Binary Search Tree equals the number of links from the root node to the deepest node.

The disadvantage of a BST is that if every item which is inserted to be next is greater than the previous item, then we will get a right skewed BST or if every item which is to be inserted is less than to the previous item, then we will get a **left skewed BST**.



So, to overcome the skewness problem in BST, the concept of AVL- tree or height balanced tree came into existence.

Balanced Binary Trees: Balancing ensures that the internal path lengths are close to the optimal n log n. A balanced tree will have the lowest possible overall height. AVL trees and B trees are balanced binary trees.

Example-1: Number of leaves in a binary search tree

```
int numberofleaves(struct bstnode * node)

{

   int total = 0;

   if(node->Left == 0 && node->Right == 0)

      return 1;

   if(node->Left!= 0)

      total += numberofleaves(node->Left);

   if(node->Right!=0)

      total += numberofleaves(node->Right);

   return total;

}
```

Example-2: Find the Diameter of a binary tree
(Diameter of a tree is the longest path between two leaf nodes in a tree.)

```
int diameter(struct btnode *root, int *height)
{
 int leftH = 0, rightH = 0;
 int leftD = 0, rightD = 0;

 if(root == NULL)
 {
 *height = 0;
 return 0;
 }
 leftD = diameter(root->left, &leftH);
 rightD = diameter(root->right, &rightH);
 *height = max(leftH, rightH) + 1;
 return max(leftH + rightH + 1, max(leftD, rightD));
}
```

Example-3: Search an element in the Binary tree using iteration

```
struct bstnode *search(int value, struct bstnode *root){
 while(root!=NULL && value!=root->value)
 {
 if(value < root->value) root = root->left;
 else root = root->right;
 }
 return(root);
}
```

Example-4: Search an item (element) in the Binary tree using Recursive function

```
struct btnode *search(int item, struct btnode *root){

 if(root==NULL || item == root->value) return(root);

 if(item < root->info) return recursive_search(item, root->left);

 else return recursive_search(item, root->right);

}
```

Example-5: Check if given binary tree is Binary Search Tree

```
bool isbst(struct btnode* root,int min,int max)

{

 if(root==NULL) return true;

 if(n->data<=min || n->data > max) return false;

 if(!isbst(root->left,min,n->data) || !isbst(root->right,n->data,max)) return false;

 return true;

}
```

Example-6:
Write a recursive function to count the number of nodes in binary tree

```
int count(stuct btnode* root) {

 if(root == NULL) return 0;

 else return count(root->left) + count(root->right) + 1;

}
```

AVL Trees
A binary search tree is an AVL tree if and only if each node in the tree satisfies the following property:

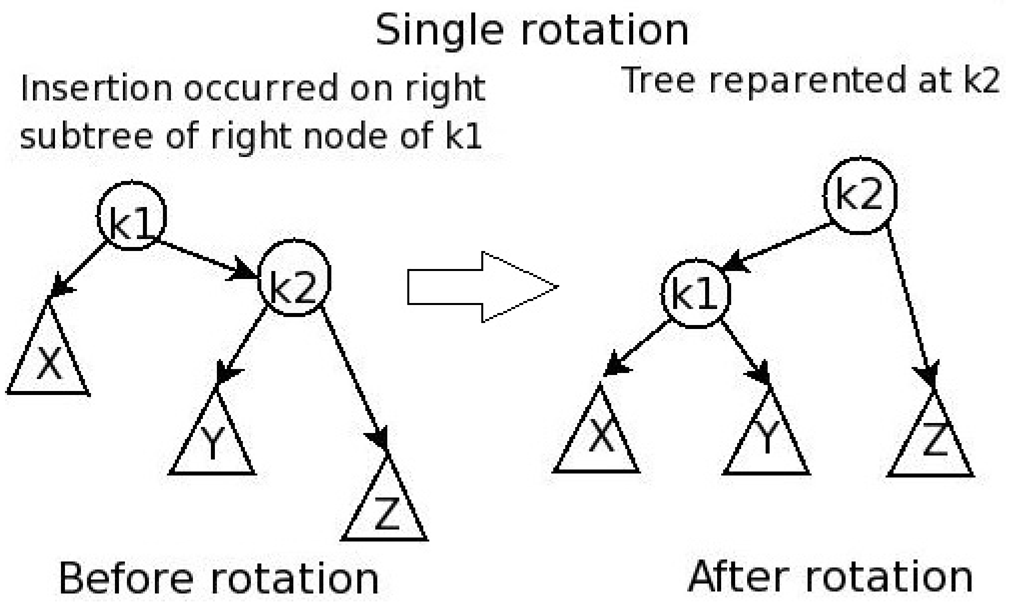
- The height of the left subtree can differ from the height of the right subtree by at most 1.
- Every subtree of the tree is an AVL tree.

An AVL (Adelson-Velskii and Landis) is a binary tree with the following properties.

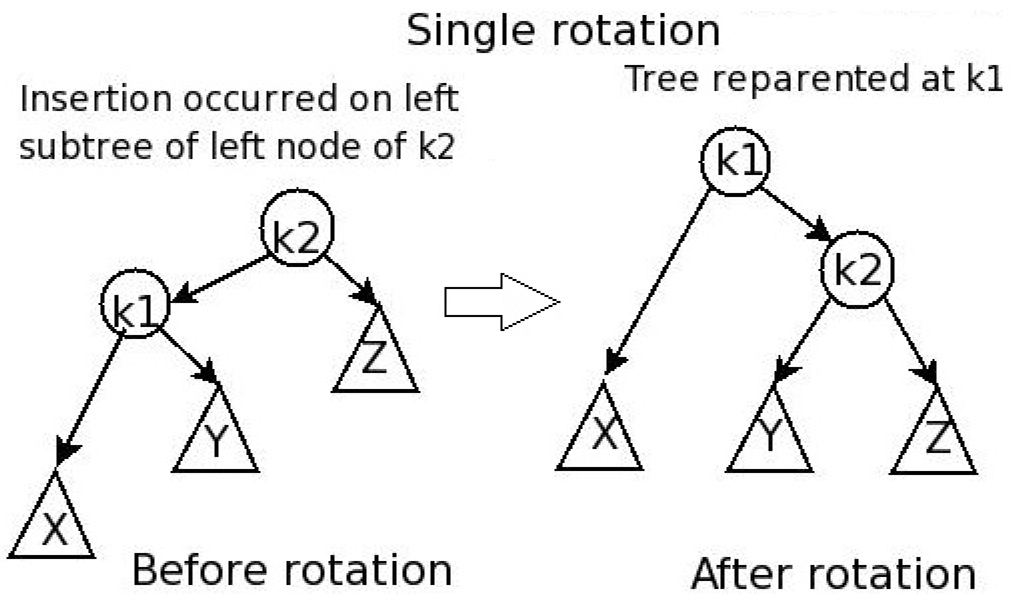
- For any node in the tree, the height of the left and right subtrees can differ by atmost 1.
- The height of an empty subtree is –1.
- Every node of an AVL tree is associated with a balance factor.
- Balance factor of a node = Height of left subtree – Height of right subtree
- A node with balance factor – 1, 0 or 1 is considered as balanced.
- AVL Tree is height balanced binary tree.
- The objective is to keep the structure of the binary tree always balanced with n given nodes so that the height never exceeds O(log n).
- After every insert or delete we must ensure that the tree is balanced.
- A search of the balanced binary tree is equivalent to a binary search of an ordered list.
- In both cases, each check eliminates half of the remaining items. Hence searching is O(log n).
- An AVL tree with n nodes has height O(log(n)).
- For search/insert/delete operations takes **worst-case** time of O(log(n)).

Rotations: A tree rotation is required when we have inserted or deleted a node which leaves the tree in an unbalanced form.

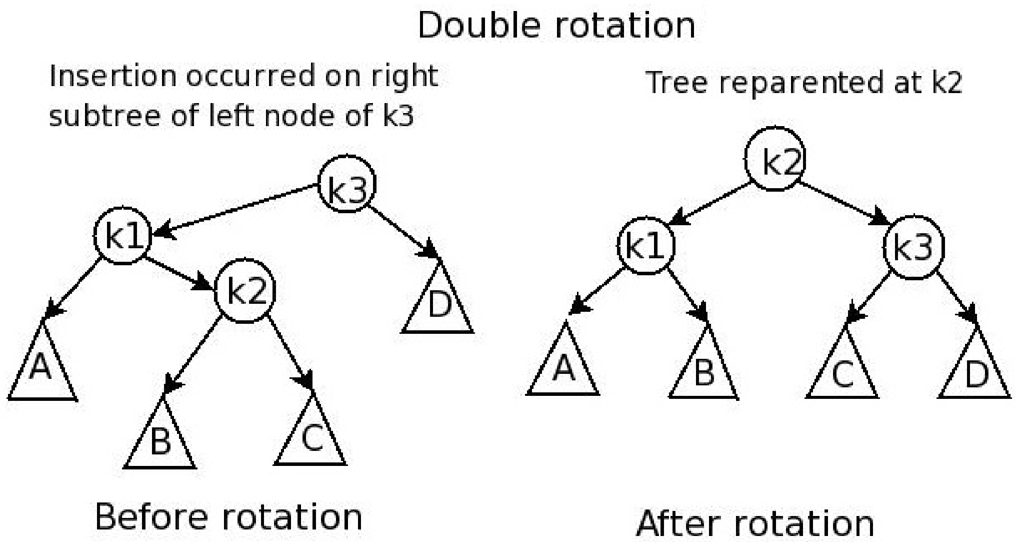
Left rotation (L-rotation): Left rotation of nodes is shown in below figure.



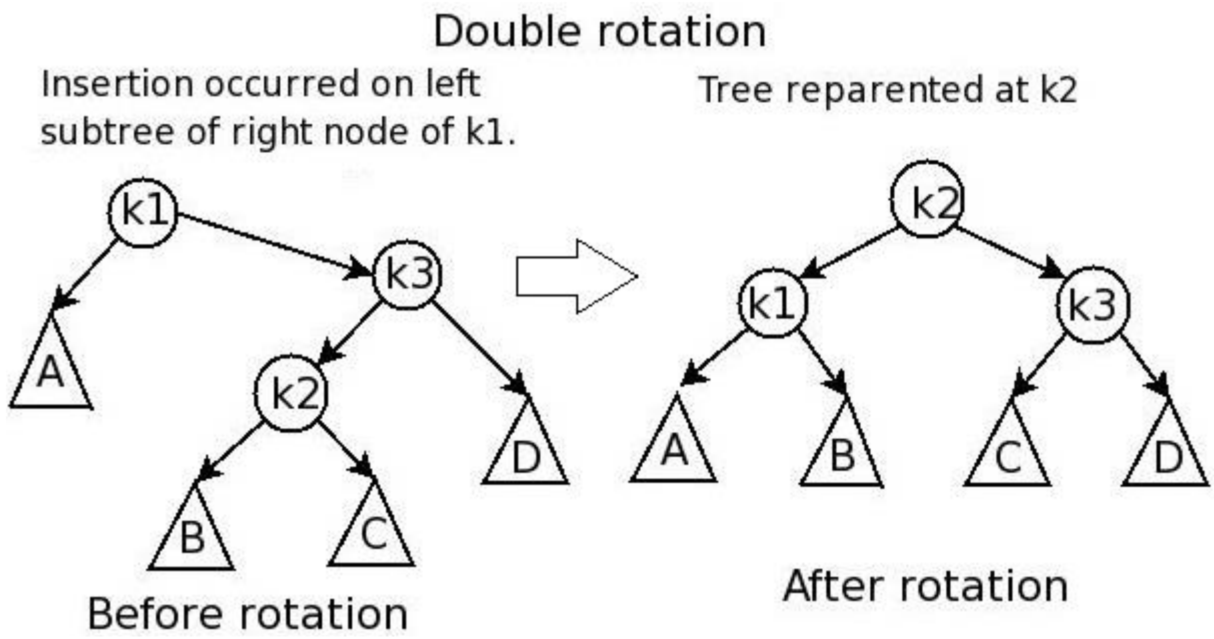
Right rotation (R-rotation): Right rotation of nodes is shown in below figure.



Double right-left rotation (R-L rotation):



Double right-left rotation (L-R rotation):



A non-empty binary tree T is an AVL-tree if and only if

$|h(T_L) - h(T_R)| \leq 1$

where, $h(T_L)$ = Height of left subtree T_L of tree T, and $h(T_R)$ = Height of right subtree T_R of tree T

$h(T_L) - h(T_R)$ is also known as Balance Factor (BF).

For an AVL (or height balanced tree), the balance factor can be either 0, 1 or –1. An AVL search tree is binary search tree which is an AVL-tree.

Insertion of a node into AVL tree

- Insert a node finding the place using BST property
- Calculate the balance factor for each node.
- Balance the AVL tree using one of four rotations if the tree is imbalanced.

Deletion of a node into AVL tree

- Delete a node with BST delete procedure.
- Balance the AVL tree using AVL rotations if the tree is imbalanced.

Minimum number of nodes in an AVL tree of height h:

Let $N(h)$ be the minimum number of nodes in a tree of height h.

$N(0)=1$, $N(1)=2$, $N(2)=4$, and

- In general, $\mathbf{N(h) = 1 + N(h-1) + N(h-2)}$

A tree with height h must have at least one child with height $h-1$, and to make the tree as small as possible, we make the other child have height $h-2$.

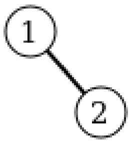
- $\mathbf{N(h) = F(h+2) - 1}$, where $F(n)$ gives the nth Fibonacci number.

Maximum number of nodes (n) in an AVL tree of height h:

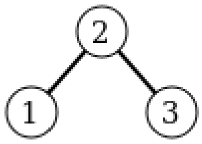
- $\mathbf{n = 2^{h+1} - 1}$

Example: Insert elements 1, 2, 3, 4, 5, 6, 7 into an empty AVL tree.

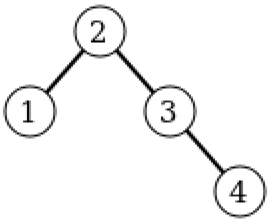
Insert 1,2:



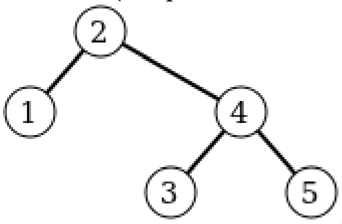
Insert 3: (requires left rotation after inserting 3 as right child to 2)



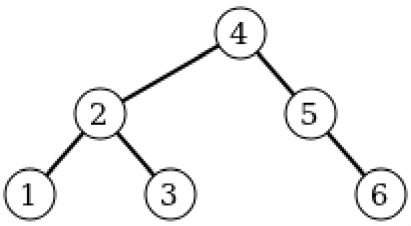
Insert 4:



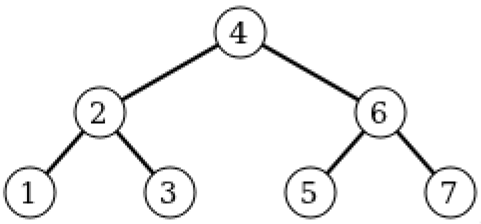
Insert 5: (Requires left rotation after inserting 5 as right child to 4)



Insert 6:



Insert 7: (Requires left rotation after inserting 7 as right child to 6)



Binary Heaps

The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest (lowest level is filled in left to right order and need not be complete). There are two types of heap trees: Max heap tree and Min heap tree.

1. **Max heap:** In a heap, for every node i other than the root, the value of a node is greater than or equal (at most) to the value of its parent. A[PARENT (i)] $\geq$ A[i]. Thus, the largest element in a heap is stored at the root.

2. **Min heap:** In a heap, for every node i other than the root, the value of a node is less than or equal (at most) to the value of its parent. A[PARENT (i)]≤ A[i]. Thus, the smallest element in a heap is stored at the root.

   The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed as follows:
   - PARENT (*i*): Parent of node *i* is at floor(*i*/2)
   - LEFT (*i*): Left child of node *i* is at 2*i*
   - RIGHT (*i*): Right child of node *i* is at (2*i* + 1)

   Since a heap is a complete binary tree, it has a smallest possible height. A heap with N nodes always has O(log N) height.

   A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

   Heapify: Heapify is a procedure for manipulating heap data structures. It is given an array A and index *i* into the array. The subtree rooted at the children of A[*i*] are heap but node A[*i*] itself may possibly violate the heap property.
   A[*i*] < A[2*i*] or A[*i*] < A[2*i* +1].

   The procedure 'Heapify' manipulates the tree rooted at A[*i*] so it becomes a heap.

   Heapify (*A*, *i*)

   l ← left [*i*]

   r ← right [*i*]

   if *l* ≤ heap-size [*A*] and *A*[*l*] > *A*[*i*]

   then largest ← *l*

   else largest ← *i*

   if *r* ≤ heap-size [*A*] and *A*[*i*] > *A*[largest]

   then largest ← *r*

   if largest ≠ *i*

   then exchange *A*[*i*] ↔ *A*[largest]

   Heapify (*A*, largest)

   Time complexity of Heapify algorithm is: O(log n)

   Building a Heap: Heapify procedure can be used in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[$\lfloor n/2 \rfloor$ +1 . . *n*] are all leaves, the

procedure Build_Heap goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

Build_Heap (A)

heap-size (*A*) ← length [*A*]

For *i* ← floor(length[*A*]/2) down to 1 do

Heapify (*A*, *i*)

Time complexity of Build_Heap algorithm is: O(n)

Heap of height *h* has the minimum number of elements when it has just one node at the lowest level.

Minimum nodes of Heap of a height h: The levels above the lowest level form a complete binary tree of height h -1 and 2^h -1 nodes. Hence the minimum number of nodes possible in a heap of height h is 2^h nodes.

Maximum nodes of Heap of a height h: Heap of height *h*, has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height *h* and hence has (2^{h+1} -1) nodes.

For Min heap tree of n-elements:
- Insertion of an element: O(*log* n)
- Delete minimum element: O(*log* n)
- Remove an element: O(*log* n)
- Find minimum element: O(1)

DecreaseKey(p,d) operation on heap:
- This operation lowers the value of the element at position p by a positive amount d.
- It is used to increase the priority of an element.
- We have to find a new position of the element according to its new priority by percolating up.

IncreaseKey(p,d) operation on heap:
- This operation increases the value of the element at position p by a positive amount d.
- It is used to decrease the priority of an element.
- We have to find a new position of the element according to its new priority by percolating down.

Remove(p) operation on heap:
- With this operation an element p is removed from the queue.
- This is done in two steps: Assigning the highest priority to p – percolate p up to the root.

- Deleting the element in the root and filling the hole by percolating down the last element in the queue.

Heap Sort:
The heap sort combines the best of both merge sort and insertion sort.
Like merge sort, the worst case time of heap sort is O(n log n) and like insertion sort, heap sort sorts in-place.

- Given an array of n element, first we build the heap.
- The largest element is at the root, but its position in sorted array should be at last. So, swap the root with the last element and heapify the tree with remaining *n*-1 elements.
- We have placed the highest element in its correct position. We left with an array of*n*-1 elements. repeat the same of these remaining *n*-1 elements to place the next largest element in its correct position.
- Repeat the above step till all elements are placed in their correct positions.

```
heapsort(A) {

 BUILD_HEAP (A)

 for (i = length (A); i>=2; i--){

 exchange (A[1], A[i]);

 heap-size [A] = heap-size [A] - 1;

 Heapify (A, 1);

 }

 }
```

Graphs

A *graph* is a collection of nodes called *vertices*, and the connections between them, called*edges*.

Directed Graph: When the edges in a graph have a direction, the graph is called a *directed graph* or *digraph* and the edges are called *directed edges* or *arcs*.

Adjacency: If (u,v) is in the edge set we say **u is adjacent to v**.

Path: Sequence of edges where every edge is connected by two vertices.

Loop: A path with the same start and end node.

Connected Graph: There exists a path between every pair of nodes, no node is disconnected.
Acyclic Graph: A graph with no cycles.

Weighted Graphs: A weighted graph is a graph, in which each edge has a weight.

Weight of a Graph: The sum of the weights of all edges.

Connected Components: In an undirected graph, a connected component is a subset of vertices that are all reachable from each other. The graph is connected if it contains exactly one connected component, i.e. every vertex is reachable from every other. Connected component is a maximal connected subgraph.

Subgraph: subset of vertices and edges forming a graph.

Tree: Connected graph without cycles.

Forest: Collection of trees

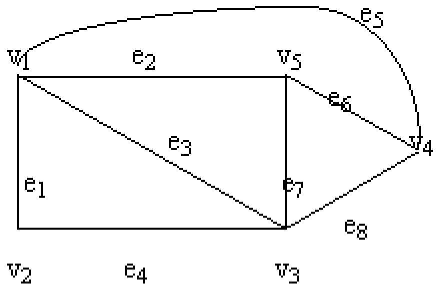
In a directed graph, a **strongly connected component** is a subset of mutually reachable vertices, i.e. there is a path between every two vertices in the set.

Weakly Connected component: If the connected graph is not strongly connected then it is weakly connected graph.

Graph Representations: There are many ways of representing a graph:

- Adjacency List
- Adjacency Matrix
- Incidence list
- Incidence matrix

Example: Consider the following graph G.



Adjacency List representation of above graph G:

- v1: v2, v3, v4, v5
- v2: v1, v3
- v3: v1, v2, v4, v5
- v4: v1, v3, v5
- v5: v1, v3, v4

Adjacency Matrix representation of above graph G:

$$\begin{array}{c|c} & \begin{matrix} v1 & v2 & v3 & v4 & v5 \end{matrix} \\ \begin{matrix} v1 \\ v2 \\ v3 \\ v4 \\ v5 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Incidence list representation of above graph G:

{(1, 2, (1, 5), (1, 3), (1, 4), (4, 5), (3, 5), (3, 4), (2, 3)}

Incidence matrix representation of above graph G:

$$\begin{array}{c|c} & \begin{matrix} e1 & e2 & e3 & e4 & e5 & e6 & e7 & e8 \end{matrix} \\ \begin{matrix} v1 \\ v2 \\ v3 \\ v4 \\ v5 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Graph Traversals: Visits all the vertices that it can reach starting at some vertex. Visits all vertices of the graph if and only if the graph is connected (effectively computing Connected Components). Traversal never visits a vertex more than once.

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph.

Depth first search (DFS) Algorithm

Step 1: Visit the first vertex, you can choose any vertex as the first vertex (if not explicitly mentioned). And push it to the Stack.

Step 2: Look at the undiscovered adjacent vertices of the top element of the stack and visit one of them (in any particular order).

Step 3: Repeat Step 2 till there is no undiscovered vertex left.

Step 4: Pop the element from the top of the Stack and Repeat Step 2, 3 and 4 till the stack is not empty.

```
DepthFirst(Vertex v){

 mark v as Visited

 for each neighbor w of v {

   if (w is not visited){

     add edge (v,w) to tree T

     DepthFirst(w)
```

```
  }
 }
}
```

Using DFS, only some edges will be traversed. These edges will form a tree, called the**depth-first-search tree** of G starting at the given root, and the edges in this tree are called**tree edges**. The other edges of G can be divided into three categories:

- **Back edges** point from a node to one of its ancestors in the DFS tree.
- **Forward edges** point from a node to one of its descendants.
- **Cross edges** point from a node to a previously visited node that is neither an ancestor nor a descendant.

Applications of DFS

- Minimum spanning tree
- To check if graph has a cycle
- Topological sorting
- To find strongly connected components of graph
- To find bridges in graph

Analysis of the DFS: The running time of the DFS algorithm is O(|V|+|E|).

Breadth First Search Algorithm

Step 1: Visit the first vertex, you can choose any node as the first node. And add it into the a queue.

Step 2: Repeat the below steps till the queue is not empty.

Step 3: Remove the head of the queue and while staying at the vertex, visit all connected vertices and add them to the queue one by one (you can choose any order to visit all the connected vertices).

Step 4: When all the connected vertices are visited. Repeat Step 3.

```
Breadth-First-Search (G) {
    initialize a queue Q
    unmark all vertices in G
    for all vertices a in G {
        if (a is unmarked) {
            enqueue (Q, a)
            while (!empty (Q) {
                b = dequeue (Q)
```

```
                    if (b is unmarked) {
                          mark b
                          visit b // print or whatever
                          for all vertices c
                                adjacent from b {
                                enqueue (Q, c)
                          }
                    }
              }
        }
    }
}
```

Applications of BFS

- To find shortest path between two nodes u and v
- To test bipartite-ness of a graph
- To find all nodes within one connected component
- To check if graph has a cycle
- Diameter of Tree

Analysis of BFS: The running time of the BFS algorithm is O(|V|+|E|).

Graph Applications:

- Electronic circuits
- Task scheduling
- Route mapping
- Packet routing in Networks

Spanning Tree

- A spanning tree of an undirected graph is a subgraph that contains all the vertices, and no cycles.
- If we add any edge to the spanning tree, it forms a cycle, and the tree becomes a graph.
- Number of nodes in the spanning tree: |V|
- Number of edges in the spanning tree: |V|-1
- Spanning Tree may not be unique.

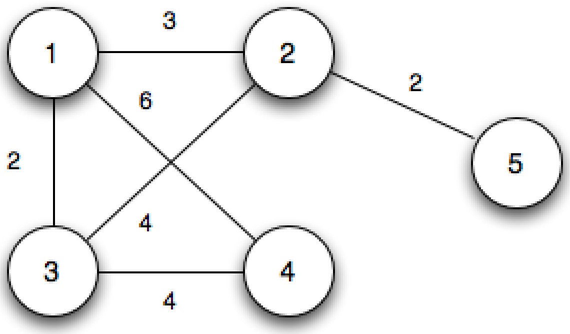
Minimum Spanning Tree (MST): A spanning tree whose weight is minimum over all spanning trees is called a minimum spanning tree.

- MST is spanning tree.
- Number of nodes in the spanning tree: |V|
- Number of edges in the spanning tree: |V|-1
- MST may not be unique.
- It has no cycles.

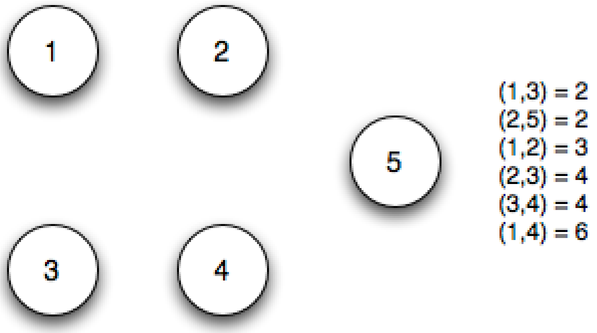
Kruskal's Algorithm

```
MST-KRUSKAL(G,w)

 A = Ø

 for each vertex v ∈ G.V

   MAKE-SET(v)

   sort edges of G.E into nondecreasing order by weight w

   for each edge (u,v) ∈ G.E, taken in nondecreasing order by weight

     if FIND-SET(u) ≠ FIND-SET(v)

        A = A ∪ {(u,v)}

        UNION(u,v)

 return A
```

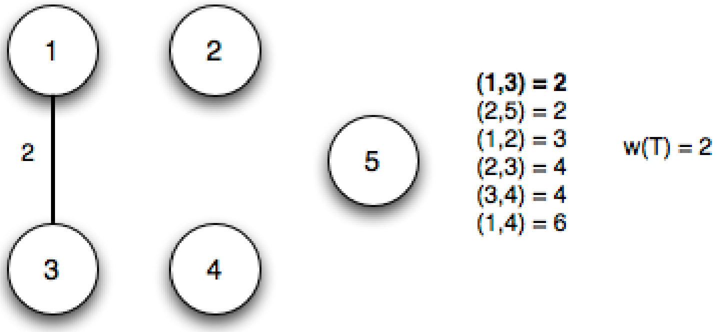
Example: Consider the following graph to computer MST using Kruskal's algorithm.



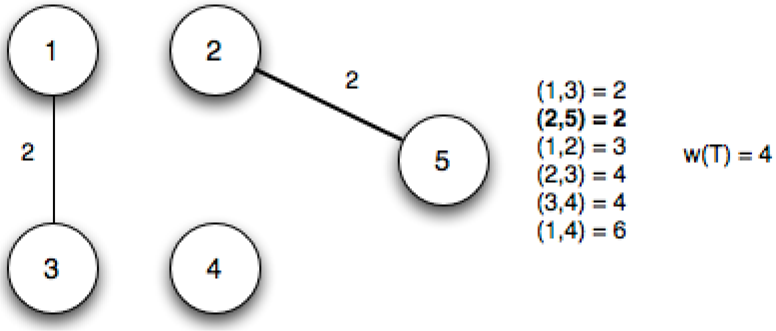
- Make each vertex a separate tree



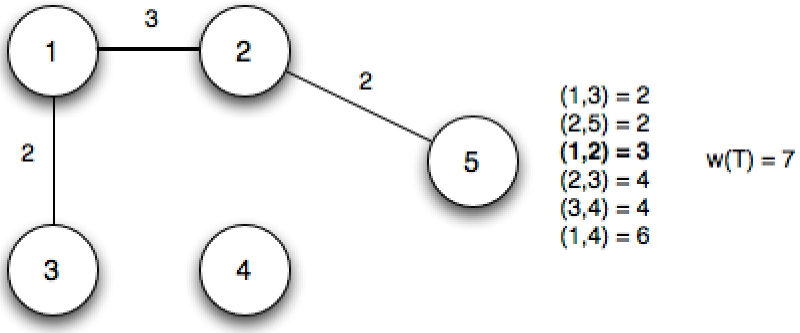
- Sort the edges in non-decreasing order and select minimum weight(cost/distance) edge. (1,3) edge has minimum weight of the graph.



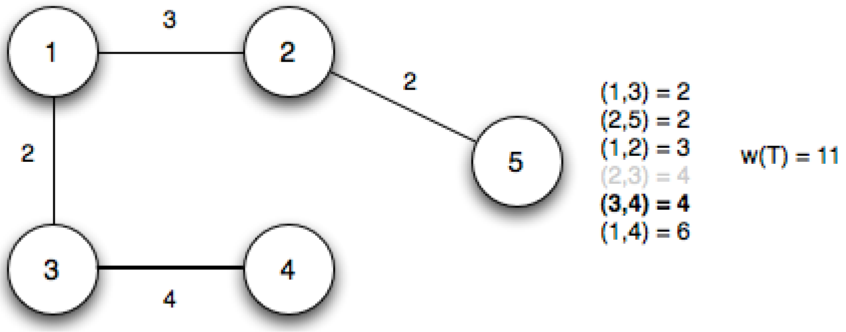
- Sort the remaining edges in non-decreasing order and select next minimum weight(cost/distance) edge. (2, 5) edge has minimum weight of the graph.



- Similarly select minimum weight edge (1,2) and add to the graph. .



- Edge (2,3) forms cycle, so add next edge (3,4) to graph.



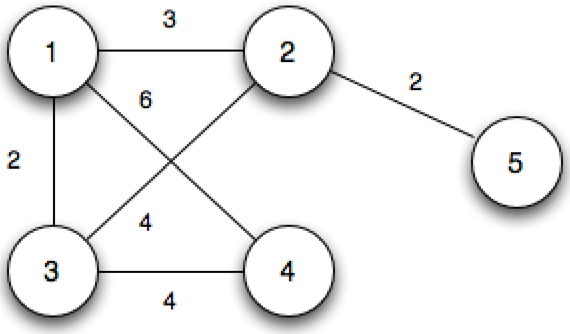
Analysis of Kruskal's Algorithm:

- Running time is O(E log V)

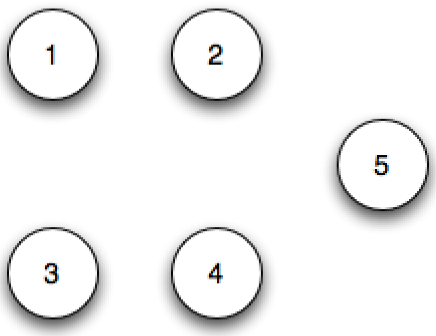
Prim's Algorithm
A ← V[G]
for each vertex u in Q {

```
key [u] ← ∞
}
key [r] ← 0
π[r] ← NIL
while array A is empty {
scan over A find the node u with smallest key, and remove it from array A
for each vertex v in Adj[u] {
if v is in A and w[u, v] < key[v] {
π[v] ← u
key[v] ← w[u, v]
}
}
}
```

Example: Consider the following graph to computer MST using Prim's algorithm.



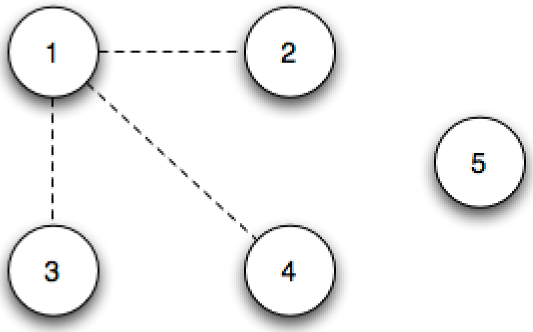
- Initialize Q with some vertex. Assume 1 is starting vertex.



Q

key	u	π
0	1	/
∞	2	/
∞	3	/
∞	4	/
∞	5	/

- Dequeue vertex 1, and update Q by changing.
  - $u_3.key = 2$ (edge (u_1, u_3)),
  - $u_2.key = 3$ (edge (u_1, u_2)),
  - $u_4.key = 6$ (edge (u_1, u_4))

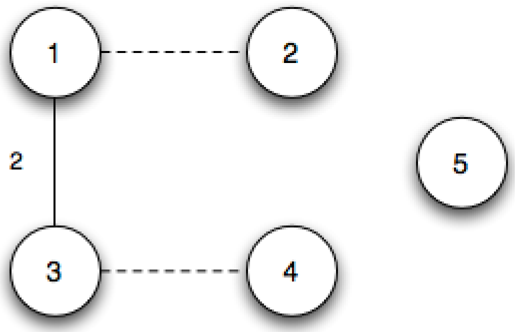


Q

key	u	π
2	**3**	**1**
3	**2**	**1**
6	**4**	**1**
∞	5	/

w(T) = 0

- Dequeue vertex 3 (adding edge (u_1, u_3) to *T*) and update *Q* by changing
  - $u_4.key$ = 4 (edge (u_3, u_4))

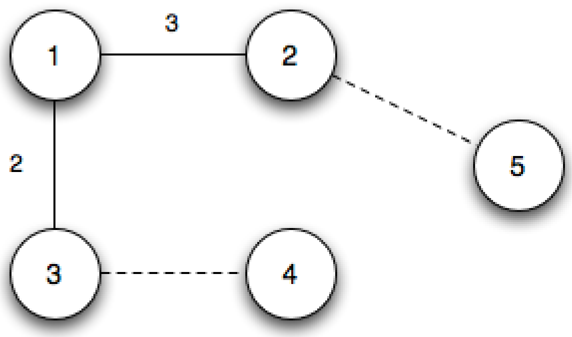


Q

key	u	π
3	2	1
4	**4**	**3**
∞	5	/

w(T) = 2

- Dequeue vertex 2 (adding edge (u_1, u_2) to *T*) and update *Q* by changing
  - $u_5.key$ = 2 (edge (u_2, u_5))

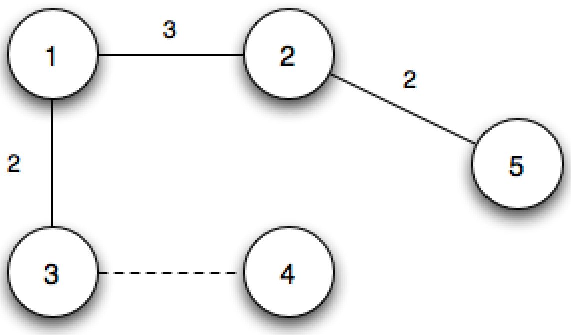


Q

key	u	π
2	**5**	**2**
4	4	3

w(T) = 5

- Dequeue vertex 5 (adding edge (u_2,u_5) to *T*) with no updates to *Q*

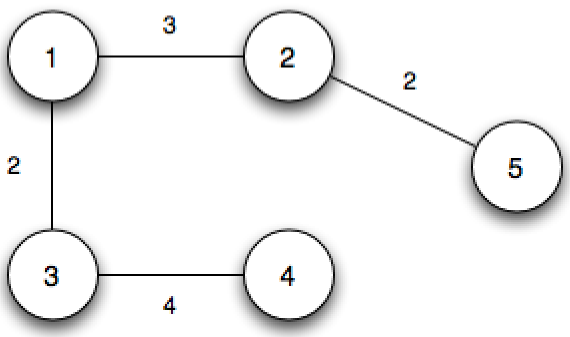


Q

key	u	π
5	4	3

w(T) = 7

- Dequeue vertex 4 (adding edge (u_3,u_4) to T) with no updates to Q

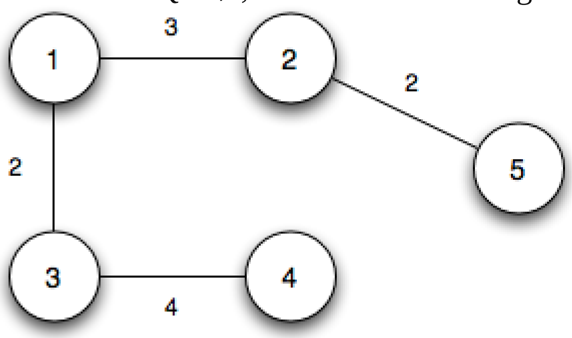


Q

key	u	π

w(T) = 11

- Now $Q = \emptyset$, So the final MST is given below with MST weight =11



w(T) = 11

Analysis of Prim's Algorithm:

- Using Adjacency list: $O(V^2)$ time
- Using Fibonacci heap: **O(E + V lg V) time**

Introduction of Algorithm

Algorithm:

- It is a finite step-by-step procedure to achieve a required result.
- It is a sequence of computational steps that transform the input into the output.

- It is a sequence of operations performed on data that have to be organized in data structures.
- It is an abstraction of a program to be executed on a physical machine.

Analysis of Algorithms:

Analysis of algorithms is the study of computer program performance and resource usage. Following things are considered to be very important to design the algorithm.

- Modularity of the program
- User-friendliness
- Correctness of the program
- Programmer's time
- Maintainability
- Security
- Functionality
- Robustness
- Simplicity
- Extensibility
- Reliability
- Scalability

Algorithm can be designed for the many problems:

- Sorting n numbers
- Searching an element
- Finding maximum and minimum element from the given list
- Sum of n numbers
- Single source shortest path
- Minimum spanning tree
- Travelling Sales Person problem
- All pairs shortest path, etc.

Strategy of Algorithms:

- Greedy Algorithms
- Divide & conquer Algorithms
- Prune & search Algorithms
- Dynamic programming
- Branch and bound Technique
- Approximation Algorithms
- Heuristics

Example-1: Maximum of the given array of elements.

```
Maximum(A, n)

{

  Max = A[0];
```

```
 for (i = 1 to n −1)

 { if Max < A[i] then Max = A[i]; }

 return Max;

}
```

Searching

There are two types of search algorithms.

- Linear Search (Sequential Search)
- Binary Search

Linear Search

- Linear Search is the simplest method to solve the searching problem.
- Linear search has no assumptions about the order of the list.
- It finds an item in a collection by looking for it from the beginning of array and looks for it till the end. It returns the first position (index) of an item whenever it finds.

Pseudo code of Sequential search:

```
int linearsearch (int a [ ], int first, int last, int key)
{
        for (int i = first; i < = last; i ++)
        {
                if (key = = a [i])
                {
                return i;        // successfully found the
                }        // key and return location
        }
        return – 1;                // failed to find key element
}
```

Analysis of Sequential Search: The time complexity in sequential search in all three cases is given below.

- **Best case:**
  - The best case occurs when the search term is in the first slot in the array.
  - Number of comparisons in best case = 1.
  - Time complexity = O(1).
- **Worst case:**
  - The worst case occurs when the search term is in the last slot in the array, or is not in the array.
  - The number of comparisons in worst case = size of the array = n.
  - Time complexity = O(n)
- **Average case:**

- On average, the search term will be somewhere in the middle of the array.
- The average number of comparisons = n/2
- Time complexity = O(n)

Binary Search

- Binary search assumes the list is already in order.
- In each step, binary search algorithm divides the array into three sections.

1. Finds the Middle element
2. Considers only left side elements of middle elements if the searching element is less than middle.
3. Considers only right side elements of middle elements if the searching element is greater than middle.

Pseudo Code of Binary Search:

```
int binarysearch (int a[ ], int n, int key)
    {
            int first = 0, last = n – 1, middle;
                    while (first < = last)
            {
                    middle = (first + last)/2;  /*
                    calculate middle* /
                    if (a [middle] == value) /*
                    if value is found at mid */
                    {
                                    return middle;
                            }
                                    else if (a [middle] > value) /*
                                    if value is at left half */
                            {
                                    last = middle – 1;
                    }
                            else
                            first = middle + 1; /*
                            if value is in right half */
            }
                            Return – 1;
    }
```

Analysis of Binary Search: The time complexity in Binary search in all three cases is given below.

- **Best case:**
  - The best case occurs when the search term is in the middle of the array.
  - Number of comparisons in best case = 1.
  - Time complexity in the best case = O(1).
- **Worst case:**
  - The worst case for binary search occurs in the following cases:
    - when the search term is not in the list, or

        - when the search term is one item away from the middle of the list, or
        - when the search term is the first or last item in the list.
    - The maximum number of comparisons in worst case = ($\log_2 n$) + 1
    - Time complexity in the worst case = O($\log_2 n$)
- **Average case:**
    - The average case occurs when the search term is anywhere else in the list.
    - Number of comparisons = O($\log_2 n$)
    - Time complexity in the average case = O($\log_2 n$)

Analysis of Binary Search: The time complexity of binary search in all three cases is given below

- **Best case:** The best case complexity is O(1)
- **Average case**: T(n) = O($\log_2 n$)
- **Worst case:** In worst case, the complexity of binary search is O($\log_2 n$)
    - The number of comparisons performed by Algorithm binary search on a sorted array of size n is at most log n + 1.

Applications of Binary Search:

- To find the first instance of an item (element).
- To find the last instance of an item (element)
- To find the number of instances of an item.
- Given an array containing only zero's and one's in sorted order. You can find the first occurrence of 1 in array.

Linear Search Vs Binary Search

- Linear search is sequential, but binary search uses divide and conquer approach
- Linear search begins the search from first position of array, whereas binary search begins from middle position of array.
- Linear search can be applied to any array, but binary search can be applied to only sorted array.
- Linear search worst case time complexity is O(n), and Binary search worst case time complexity is O($\log_2 n$).
- After k th comparison, number of remaining elements left for searching in Linear search is (n-i), and in Binary search is $n/(2^k)$ approximately.

Example-1: Recursive implementation for Binary Search

```
int binarySearch(int a[ ], int start, int end, int key){

if(start <= end) {

 int mid = (start + end)/2;

 if(a[mid] == key) return mid;

 if(a[mid] < key)
```

```
  return binarySearch(a, mid+1, end, key) ;
 else
  return binarySearch(a, start, mid–1, key) ;
 }
return –1;
```

Sorting

Sorting is ordering a list of elements.

Types of sorting: There are many types of algorithms exist based on the following criteria:

- Based on Complexity
- Based on Memory usage (Internal & External Sorting)
- Based on recursive/non-recursive implementation
- Based on stability
- Based on comparison/non-comparison, etc.

Internal Sorting: If the number of elements is small enough to fits into the main memory, sorting is called internal sorting. Bucket sort, Bubble sort, Insertion sort, Selection sort, Heap sort, and Merge sort are internal sorting algorithms.

External Sorting: If the number of elements is so large that some of them reside on external storage during the sort, it is called external sorting. External merge sort, and shell sort (Bucket sort) are external sorting algorithms.

In-place Sorting: The in-place sorting algorithm does not use extra storage to sort the elements of a list. Insertion sort, quick sort and Selection sort are in-place sorting algorithms.

Stable Sorting: Stable sorting algorithm maintain the relative order of records with equal values during sorting. Merge sort, Insertion sort, and Bubble sort are stable sorting algorithms.

Comparison based sorting: It determines which of two elements being compared should occur first in the final sorted list.

- **Exchanging:** used by Bubble Sort
- **Selection:** used by Selection Sort and Heapsort
- **Insertion:** used by Insertion Sort and Shell Sort
- **Merging:** used by Merge Sort
- **Partitioning:** used by Quick Sort

Non-Comparison based sorting: Bucket sort, Radix sort, and Counting sort are non-comparison based algorithms.

Some of the sorting algorithms are explained below.

Bubble Sort

Working Principle of Bubble Sort:

It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

It is comparison based sorting algorithm (It uses only comparisons to sort the elements).

Pseudo Code for Bubble Sorting:

```
void BubbleSort (int a[ ], int n)
    {
            int i, temp, j;
            for (i = 1; i < = n; i ++)
            {
                    for (j = 1; j < = i; j++)
                    {
                            if (a[j] > a[j + 1])
                            {
                                    temp = a [j]
                                    a[j] = a[j + 1];
                                    a[j + 1] = temp;
                            }
                    }
              }
    }
```

Analysis of Bubble Sort: The time complexity of bubble sort in all three cases is given below

- Best case Complexity: $O(n^2)$
- Average case Complexity: $O(n^2)$
- Worst case Complexity: $O(n^2)$

Insertion Sorting

Working Principle of Insertion Sort:
Every iteration of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The resulting array after i iterations has the property where the first i+1 entries are sorted.
The insertion sort only passes through the array once. Therefore, it is very fast and efficient sorting algorithm with small arrays. Insertion sort is useful only for small files or very nearly sorted files.

It is comparison based, in-place and stable algorithm.

Pseudo Code of Insertion Sort:

```
int insertionsort (int a[ ], int n)
{
        int i, j, key;
        for (j = 2; j < = n; j++)
        {
                key = a [j]
                i = j – 1;
                while (i > 0 && a [i] > key)
                {
                        a[i + 1] = a [i];
                        i = i – 1;
                }
                a [i + 1] = key;
        }
}
```

Analysis of Insertion Sort

- **Best case complexity:** Array is already sorted.
  - $T(n) = O(n)$
- **Average case complexity:** All permutations are equally likely.
  - $T(n) = O(n^2)$
- **Worst case complexity:** Input is in reverse sorted order.
  - $T(n) = O(n^2)$

Selection Sorting

Working Principle of Selection Sort:

- Find the minimum element of an array of n elements. Swap it with the value in the first position of array. Next, we find the minimum of the remaining $n - 1$ elements and swap with second position of array. We continue this way until the second largest element is stored in $A[n-1]$.
- Selection sort is an in-place comparison sorting algorithms.

Pseudo Code of Selection Sort:

Input: An array A[1…n] of n elements.

Output: A[1…n] sorted in non-decreasing order.

for(i=1; i<n; i++)

```
{

k=i;

for(j=i+1; j<n; j++)

{

if(A[j] < A[k] then k=j;

}

if(k ≠ i) then swap(A[i], A[k]);

}
```

Analysis of Selection Sort:

- **Worst case time complexity:** O(n^2).
- **Best case time complexity:** O(n^2).
- **Average case time complexity:** O(n^2).

Heap Sort

Heap sort is simple to implement and is a comparison based sorting. It is in-place sorting but not a stable sort.

Max heap: A heap in which the parent has a larger key than the child's is called a max heap.

Min heap: A heap in which the parent has a smaller key than the child's is called a min heap.

BUILD_HEAP (A)
heap-size (A) ← length [A]
For i ← floor(length[A]/2) down to 1 do
Heapify (A, i)

Pseudo Code of Heap Sort:

Heapify (A, i)

l ← left [i]
r ← right [i]
if $l \leq$ heap-size [A] and $A[l] > A[i]$
then largest ← l
else largest ← i
if $r \leq$ heap-size [A] and $A[i] > A$[largest]
then largest ← r
if largest ≠ i
then exchange A[i] ↔ A[largest]
Heapify (A, largest)

Heapsort(A)

1. BUILD_HEAP (*A*)
2. for *i* ← length (A) down to 2 do
1. exchange *A*[1] ↔ *A*[i]
2. heap-size [*A*] ← heap-size [*A*] – 1
3. Heapify (*A*, 1)

Analysis of Heap Sort: The total time for heap sort is O (n log n) in all three cases (best, worst and average).

- **Heapify:** which runs in O(log*n*) time.
- **Build-Heap:** which runs in linear time O(*n*).
- **Heap Sort:** which runs in O(n log*n*) time.
- **Extract-Max:** which runs in O(log*n*) time.

Merge Sort

Working principle of Merge Sort:

- Divide the unsorted list into two sublists of about half the size.
- Sort each sublist recursively by re-applying merge sort.
- Merge the two sublists back into one sorted list.

Pseudo Code for Merge Sort:

```
void merge(int a[], int low, int mid, int high)

{

int b[100];

int i = low, j = mid + 1, k = 0;



while (i <= mid && j <= high) {

if (a[i] <= a[j])

b[k++] = a[i++];

else

b[k++] = a[j++];

}

while (i <= mid)
```

```
b[k++] = a[i++];

while (j <= high)

b[k++] = a[j++];

k--;

while (k >= 0) {

a[low + k] = b[k];

k--;

}

}

-------------------------------

void mergesort(int a[], int low, int high)

{

if (low < high) {

int m = (high + low)/2;

mergesort(a, low, m);

mergesort(a, m + 1, high);

merge(a, low, m, high);

}

}

--------------------------------
```

Analysis of Merge Sort:

- **Best case time complexity:** O(n log n).
- **Average case time complexity:** O(n log n).
- **Worst case time complexity:** O(n log n).

Quick Sort

Working principle of Quick Sort:

- Select pivot element from the given list of elements.

- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way).
- After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion are lists of size zero or one, which are always sorted.
- Quick sort is comparison based, and in-place based sorting algorithm.

Pseudo Code for Quick Sort:

```
void quicksort(int *array, int start, int stop)

{

int left = start,

right = stop,

center = array[(start + stop) / 2];

while(left<right)

{

while(array[left]<center) left++;

while(array[right]>center) right--;

if(left<=right)

{

swap(&array[left], &array[right]);

left++;

right--;

}

}

if(right>start) quicksort(array, start, right);

if(left<stop) quicksort(array, left, stop);

}
```

```
void quicksort_start(int *array, int num)

{

quicksort(array, 0, num-1);

}
```

Recursive Implementation of Quick sort:

```
Partition(A, p, q) {

x = A[p];

i = p;

for (j = p+1 to q)

{

if (A[j] <= x) i = i+1;

swap(A[i], A[j]);

}

swap(A[p], A[i]);

return i;

}

Quicksort(A, p, r) // sorts list A[p..r]

{

if (p < r)

{q = Partition(A, p, r);

Quicksort(A, p, q-1);

Quicksort(A, q+1, r);

}

}
```

Analysis of Quick Sort:

- **Best case time complexity:** O(n log n).

- **Average case time complexity:** O(n log n).
- **Worst case time complexity:** O(n^2).

Note:
- Merge sort is a fast sorting algorithm whose best, worst, and average case complexity are all in O(n log n), but unfortunately it uses O(n) extra space to do its work.
- Quicksort has best and average case complexity in O(n log n), but unfortunately its worst case complexity is in O(n^2).

Hashing

Hashing is a common method of accessing data records using the hash table. Hashing can be used to build, search, or delete from a table.

Hash Table: A hash table is a data structure that stores records in an array, called a hash table. Hash table can be used for quick insertion and searching.

Load Factor: The ratio of the *number of items in a table* to *the table's size* is called the *load factor*.

Hash Function:

- It is a method for computing table index from key.
- A good hash function is simple, so it can be computed quickly.
- The major advantage of hash tables is their speed.
- If the hash function is slow, this speed will be degraded.
- The purpose of a hash function is to take a range of key values and transform them into index values in such a way that the key values are distributed randomly across all the indices of the hash table.
- Goal of hash function:The probability of any two keys hashing to the same slot is 1/N.

There are many hash functions approaches as follows:

Division Method:

- Mapping a key K into one of m slots by taking the remainder of K divided by m.

  $$h(K) = K \bmod m$$

- Example: Assume a table has 8 slots (m=8). Using division method, insert the following elements into the hash table. 36, 18, 72, 43, and 6 are inserted in the order.

36 % 8 = 4

18 % 8 = 2

72 % 8 = 0

43 % 8 = 3

6 % 8 = 6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36		6	

Mid-Square Method: Mapping a key K into one of m slots, by getting the some middle digits from value K^2.

h(k) = K2 and get middle ($\log_{10}$ m) digits

Example: 3121 is a key and square of 3121 is 9740641. Middle part is 406 (with a table size of 1000)

Folding Method: Divide the key K into some sections, besides the last section, have same length. Then, add these sections together.

- Shift folding (123456 is folded as 12+34+56)
- Folding at the boundaries (123456 is folded as 12+43+56)

Radix Hashing: Transform the number into another base and then divide by the maximum address.

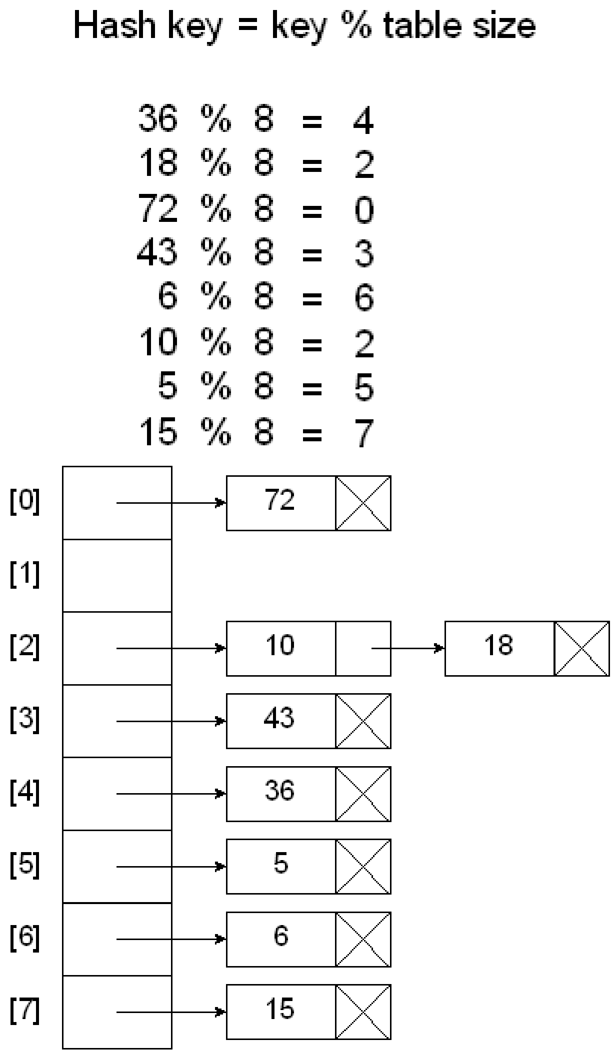
- **Example:** Addresses from 0 to 99, key = 453 in base 11 = 382. Hash address = 382 mod 99 = 85.

Problems with hashing:

- **Collision:** No matter what the hash function, there is the possibility that two different keys could resolve to the same hash address. This situation is known as a collision.
- **Handling the Collisions:** The following techniques can be used to handle the collisions.
    - Chaining
    - Double hashing (Re-hashing)
    - Open Addressing (Linear probing, Quadratic probing, Random probing), etc.

Chaining:

- A chain is simply a linked list of all the elements with the same hash key.
- A linked list is created at each index in the hash table.
- **Hash Function:** h (K) = K mod m
- **Example:** Assume a table has 8 slots (m=8). Using the chaining, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.



- A data item’s key is hashed to the index in simple hashing, and the item is inserted into the linked list at that index.
- Other items that hash to the same index are simply added to the linked list.
- Cost is proportional to length of list.
- Average length = N / M, where N is size of array and M is the number of linked lists.
- **Theorem:** Let α = N / M > 1 be average length of list. For any t > 1, probability that list length > t α is exponentially small in t.
- **Analysis of Chaining:**
    - If M is too large, then too many empty chains.
    - If M is too small, then chains are too long.
- There is no need to search for empty cells in the array or table.
- **Advantages of Chaining:** Unbounded elements can be stored (No bound to the size of table).
- **Disadvantage of Chaining:** Too many linked lists (overhead of linked lists)

Open Addressing: In open addressing, when a data item can’t be placed at the hashed index value, another location in the array is sought. We’ll explore three methods of open addressing, which vary in the method used to find the next empty location. These methods are linear probing, quadratic probing, and double hashing.

Linear Probing:

- When using a linear probing method the item will be stored in the **next available slot**in the table, assuming that the table is not already full.
- This is implemented via a linear searching for an empty slot, from the point of collision.
- If the end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.
- Example: Assume a table has 8 slots (m=8). Using Linear probing, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.

Hash key = key % table size

36 % 8 = 4
18 % 8 = 2
72 % 8 = 0
43 % 8 = 3
6 % 8 = 6
10 % 8 = 2
5 % 8 = 5
15 % 8 = 7

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

- Relationship between probe length (P) and load factor (L) for linear probing:
  - For a successful search: $P = (1 + 1 / (1–L)^2) / 2$
  - For an unsuccessful search: $P = (1 + 1 / (1–L)) / 2$
- **Analysis of Linear Probing:**
  - If load factor is too small then too many empty cells.

Quadratic Probing:

If the collision occurs, instead of moving one cell, move i^2 cells from the point of collision, where i is the number of attempts to resolve the collision.

Example:

- **Example:** Assume a table has 10 slots. Using Quadratic probing, insert the following elements in the given order.

  89, 18, 49, 58 and 69 are inserted into a hash table.

89 % 10 = 9

18 % 10 = 8

49 % 10 = 9

49 = 9 is occupied, so 1 attempt needed. Insert at $9 + 1^2 = 10 = 0$ location.

58 % 10 = 8 3 attempts – $3^2 = 9$ spots

58 = 8 is occupied, and next 3 locations are occupied.

So $8 + 3^2 = 17 = 7$ location.

69 % 10 = 9 2 attempts – $2^2 = 4$ spots

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

- **Advantages of Quadratic probing:** Compared to linear probing access becomes inefficient at a higher load factor.
- **Disadvantages of Quadratic probing:** Insertion sometimes fails although the table still has free fields.

Double Hashing:

- Double hashing requires that the size of the hash table is a prime number.
- Double hashing uses the idea of applying a second hash function to the key when a collision occurs.
- The primary hash function determines the home address. If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N). This c is added to the home address to produce an overflow addresses: if occupied, proceed by adding c to the overflow address, until an empty slot is found.
- Primary hash function is similar to direct hashing. Hash1(key) = Key % table size. A popular secondary hash function is: $Hash_2$(key) = R – (key % R) where R is a prime number that is smaller than the size of the table.
- **Example:** Assume a table has 10 slots. Primary hash function is H1(key) = key mod 10, and secondary hash function is H2(key)=7-(key mod 7). With Double hashing, insert the following elements in the given order.

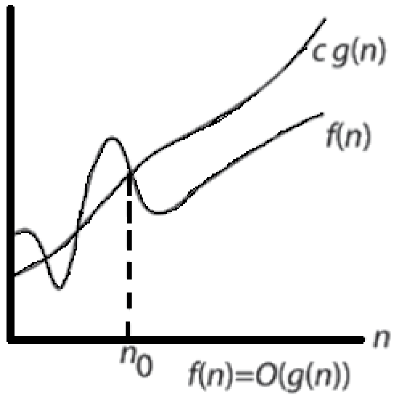
  89, 18, 49, 58 and 69 are inserted into a hash table

Space and Time Complexity

Asymptotic Notations: The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2 \ldots \}$. The asymptotic notations consist of the following useful notations.

Big Oh (O):

- If we write f(n) = O(g(n)), then there exists a function f(n) such that $\forall\ n \geq n_0$, f(n) $\leq$ cg (n) with any constant c and a positive integer n_0. **Or**
- f(n) = O(g(n)) means we can say g(n) is an asymptotic upper bound for f(n).



- *If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(max(g_1(n), g_2(n)))$*
- Transitive Property: If $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$.
- **Example-1:** Let f(n) = $n^2 + n + 5$. Then f(n) is O(n^2), and f(n) is O(n^3), but f(n) is not O(n).
- **Example-2:** Let f(n) = 3^n . Then f(n) is O(4^n) but not f(n) is not O(2^n)

Note: $O(1)$ refers to constant time. $O(n)$ indicates linear time; $O(n^k)$ (k fixed) refers to polynomial time; $O(\log n)$ is called logarithmic time; $O(2^n)$ refers to exponential time, etc.

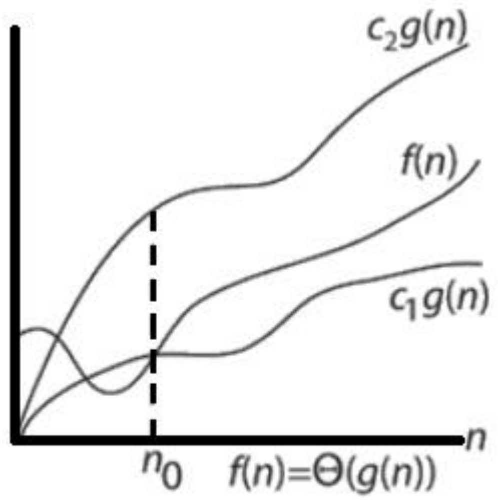
$$O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$$

Big Omega (Ω):

- If we write f(n) = Ω(g(n)), then there exists a function f(n) such that $\forall\ n \geq n_0,$ f(n) $\geq$ cg(n) with any constant c and a positive integer n_0. **Or**
- f(n) = Ω(g(n)) means we can say Function g(n) is an asymptotic lower bound for f(n).
- **Example-1:** Let $f(n) = 2n^2 + 4n + 10$. Then $f(n)$ *is* $\Omega(n^2)$

Big Theta (θ):

- If we write f(n) = θ(g(n)), then there exists a function f(n) such that $\forall\ n \geq n_0,$ c_1g(n) $\leq$ f(n) $\leq$ c_2g(n) with a positive integer n_0, any positive constants c_1 and c_2. **Or**
- f(n) = θ(g(n)) means we can say Function g(n) is an asymptotically tight bound for f(n).



- f(n) = θ(g(n)) if and only if f = O(g(n)) and f(n) = Ω(g(n)).
- **Example-1:** Let $f(n) = 2n^2 + 4n + 10$. Then $f(n)$ *is* $\theta(n^2)$

Small Oh (o):

- If we write f(n) = o(g(n), then there exists a function such that f(n) < c g(n) with any positive constant c and a positive integer n_0. **Or**
- f(n) = o(g(n) means we can say Function g(n) is an asymptotically tight upper bound of f(n).
- Example: $n^{1.99} = o(n^2)$

Small Omega (ω):

- If we write f(n) = ω(g(n)), then these exists a function such that f(n) > cg(n) with any positive constant c and a positive integer n_0. **Or**
- f(n) = ω(g(n)) means we can say g(n) is asymptotically tight lower bound of f(n).
- Example: $n^{2.00001} = \omega(n^2)$ and $n^2 \neq \omega(n^2)$

Relationship between asymptotic notations:

Properties of Asymptotic notations

- **Reflexive Property:**

$$f(n) \in \Theta(g(n)) \ is\, equivalent\, to\, g(n) \in \Theta(f(n))$$

$$f(n) \sim g(n) \ is\, equivalent\, to \ g(n) \sim f(n)$$

$$f(n) \in o(g(n)) \ \ implies \ \ g(n) \notin o(f(n))$$

$$f(n) \in \omega(g(n)) \ \ implies \ \ g(n) \notin o(f(n))$$

- **Symmetric Property:**

$$f(n) \in \Theta(g(n)) \ is\, equivalent\, to\, g(n) \in \Theta(f(n))$$

$$f(n) \sim g(n) \ is\, equivalent\, to \ g(n) \sim f(n)$$

$$f(n) \in o(g(n)) \ \ implies \ \ g(n) \notin o(f(n))$$

$$f(n) \in \omega(g(n)) \ \ implies \ \ g(n) \notin \omega(f(n))$$

- **Transitive Property:**

If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$

If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$ then $f(n) \in \Omega(h(n))$

If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$ then $f(n) \in \Theta(h(n))$

If $f(n) \in o(g(n))$ and $g(n) \in o(h(n))$ then $f(n) \in o(h(n))$

If $f(n) \in \omega(g(n))$ and $g(n) \in \omega(h(n))$ then $f(n) \in \omega(h(n))$

If $f(n) \sim g(n)$ and $g(n) \sim h(n)$ then $f(n) \sim (h(n))$

Analysis of Algorithms

The *complexity* of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function.

- Algorithm can be classified by the amount of time they need to complete compared to their input size.
- The analysis of an algorithm focuses on the complexity of algorithm which depends on time or space.

There are two main complexity measures of the efficiency of an algorithm:

1. Time Complexity: The time complexity is a function that gives the amount of time required by an algorithm to run to completion.

- **Worst case time complexity:** It is the function defined by the maximum amount of time needed by an algorithm for an input of size n.
- **Average case time complexity**: The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences.
- **Amortized Running Time:** It is the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.
- **Best case time complexity:** It is the minimum amount of time that an algorithm requires for an input of size n.

2. Space Complexity: The space complexity is a function that gives the amount of space required by an algorithm to run to completion.

Worst case Time Complexities for popular data structures:

Data Structure	Worst Case Time Complexity			
	Access	Search	Insertions	Delete
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Doubly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Time Complexities for popular sorting algorithms:

Sorting Algorithms	Best Case	Average Case	Worst Case
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$

Recurrence Relations

A recurrence is a function defined in terms of One or more base cases and Itself with smaller arguments.

Example:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Above recurrence relation can be computed asymptotically that is $T(n) = O(n^2)$.

In algorithm analysis, we usually express both the recurrence and its solution using asymptotic notation.

Methods to Solve the Recurrence Relation

There are two methods to solve the recurrence relation given as: Substitution method and Master method.

1. Guess and Test Method: *There are two steps in this method*

- Guess the solution
- Use induction to find the constants and show that the solution works.

2. Master Method: The master method gives us a quick way to find solutions to recurrence relations of the form T(n) = aT (n/b) + f(n). Where, a and b are constants, $a \geq 1$ and $b > 1$)

- $T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d),\ d \geq 0$
  - *Case-1:* If $a < b^d$, $T(n) \in \Theta(n^d)$
  - *Case-2:* If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
  - *Case-3:* If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$
- *Examples:*
  - $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \boldsymbol{\Theta(n^2)}$
  - $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \boldsymbol{\Theta(n^2 \log n)}$
  - $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \boldsymbol{\Theta(n^3)}$

3. Iterative Substitution Method: Recurrence equation is substituted itself to find the final generalized form of the recurrence equation.

T(N) = 2T(N/2)+bN

Here T(N/2) is substituted with 2T((N/2)/2)+b(N/2)

T(N) = 4T(N/4)+2bN (Similarly substitute for T(N/4)

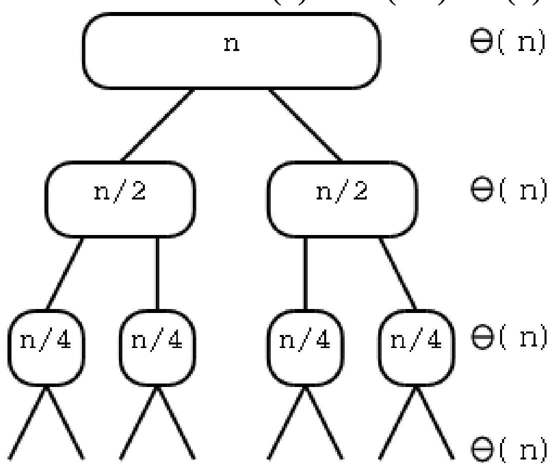
T(N) = 8T(N/8)+3bN

After (i-1) substititions,

$T(N) = 2^i T(N/2^i)+ibN$

When i=log(N), $N/2^i=1$ and we have the base case.
$T(N) = 2^{\log(N)} T(N/2^{\log(N)})+ibN$
T(N) = N T(1) +log(N)bN
T(N) = Nb +log(N)bN
Therefore T(N) is O(Nlog(N))

4. Recursion Tree Method: Using recursion method, n element problem can be further divided into two or more sub problems. In the following figure, given problem with n elements is divided into two equal sub problems of size n/2. For each level of the tree the number of elements is N. When the tree is split so evenly the sizes of all the nodes on each level. Maximum depth of tree is logN (number of levels).
Recurrence relation T(n) = 2 T(n/2) + O(n) = Θ(Nlog(N)).



Example-1: Find the Time complexity for T(N) = 4T(N/2)+N.
Solution:
Let T(N) = aT(N/b)+f(N).
Then a=4, b=2, and f(N)=N
$N^{\log_b a} = N^{\log_2 4} = N^2$.
f(N) is smaller than $N^{\log_b a}$
Using case 1 of master theorem with ε=1.
$T(N) = \Theta(N^{\log_b a}) = \Theta(N^2)$.

Example-2: Find the Time complexity for T(N) = 2T(N/2) = Nlog(N)
a=2, b=2, and f(N)=Nlog(N)
Using case 2 of the master theorem with k=1.
$T(N) = \Theta(N(\log N)^2)$.

Example-3: Find the Time complexity for T(N) = T(N/4) + 2N
a=1, b=4, and f(N)=2N
$\log_b a = \log_4 1 = 0$
$N^{\log_b} = N^0 = 1$
Using case 3: f(N) is $\Omega(N^{0+\varepsilon})$ for &epsilon=1 and af(N/b) = (1)2(N/4) = N/2 = (1/4)f(N)
Therefore $T(N) = \Theta(N)$

Example-4: Find the Time complexity for $T(N) = 9T(N/3) + N^{2.5}$
a=9, b=3, $f(N)=N^{2.5}$
$\log_b a = 2$, so $N^{\log_b} = N^2$
Using case 3 since f(N) is $\Omega(N^{0+\varepsilon})$, epsilon=.5 and $af(N/b)=9f(N/3)= 9(N/3)^{2.5}=(1/3)^{0.5}f(n)$.
Therefore T(n) is $O(N^{2.5})$

$H_1(89) = 89 \% 10 = 9$ (primary hashing)

$H_1(18) = 18 \% 10 = 8$

$H_1(49) = 49 \% 10 = 9$ a collision (primary hashing)

$H_2(49) = 7 - (49 \% 7)$ (Secondary hashing)
$= 7$ positions from [9]

$H_1(58) = 58 \% 10 = 8$

$H_2(58) = 7 - (58 \% 7)$
$= 5$ positions from [8]

$H_1(69) = 69 \% 10 = 9$

$H_2(69) = 7 - (69 \% 7)$
$= 1$ position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

- **Advantages of Double hashing:**
  - Compared to linear probing access becomes inefficient at a higher load factor.
  - Resolution sequences for different elements are different even if the first hash function hashes the elements to the same field.
  - If the hash functions are chosen appropriately, insertion never fails if the table has at least one free field.

Divide and Conquer Design Technique: Divides the problem into smaller but similar sub problems (**divide**), solve it (**conquer**), and (**combine**) these solutions to create a solution to the original problem.

- **Divide:** Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- **Conquer:** Solve the sub-problem recursively (successively and independently). If the sub problems are small enough, solve them in brute force fashion.
- **Combine:** Combine these solutions to sub-problems to create a solution to the original problem.

Divide & Conquer Algorithms:

- Mergesort
- Quicksort
- Binary tree traversals
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

Merge Sort

- Merge sort is a comparison based sorting algorithm uses divide and conquer approach.
- Merge sort is a stable sort.
- Working Principle:
  - Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining.
- **Input:** An array A[1…m] of elements and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the sub arrays A[p…q] and A[q+1…r] are sorted individually in non-decreasing order
- **Output:** A[p…r] contains the result of merging the two subarrays A[p…q] and A[q+1..r].

```
mergesort( int A[ ], int p, int r)

{

 if (p < r)

 {

 q = p + (r - p)/2;

 mergesort(A, p, q);  /* T(n/2) */

 mergesort(A, q+1, r); /* T(n/2) */
```

```
 merge(A, p, q, r); /* O(n) time */

 }

}
```

Let T(n) be the worst-case number of comparisons and data movements for an array (of size n).

a. Recursive call on the two halves of A: 2 * T(n/2)
b. Time to Merge two halves of array A : O(n)

T(n) = 2T(n/2)+ O(n)

- **Best case Time Complexity:** O(n log n)
- **Average caseComplexity:** O(n log n)
- **Worst case Complexity:** O(n log n)

Quick Sort

- It is in-place sorting.
- It is also known as partition exchange sort.
- The elements A[low..high] to be sorted are rearranged using partition.
- Pivot element which is always occupies its correct position, say A[w].
- All elements that are less than or equal to pivot occupy the positions A[low..w − 1].
- All elements that are greater than A[w] occupy the positions A[w + 1..high].
- The subarrays A[low..w − 1] and A[w+1..high] are then recursively sorted to produce the entire sorted array.

```
quicksort(A, low, high)

{

if (low < high)

{

partition( A[low...high], w);

// w is the new position of A[low];

quicksort(A, low, w-1);

quicksort(A,w+1,high);

}

}
```

Analysis of Quick Sort:

The time to sort the array of n elements is equal to

- the time to sort the left partition with **i** elements, plus

- the time to sort the right partition with **n-i-1** elements, plus
- the time to build the partitions

$$T(n) = T(i) + T(n - i - 1) + O(n)$$

- **Worst case Analysis:** It happens when the pivot is the smallest (or the largest) element.
    - Worst case Time complexity: $T(n) = O(n^2)$
- **Best case Analysis:** The pivot is in the middle and the subarrays divide into balanced partition every time.
    - Best case Time complexity: $T(n) = O(n \log n)$
- **Average case Analysis:** $O(n \log n)$.

Advantages of Quick Sort Method

- One of the fastest algorithms on average.
- It does not need additional memory, so this is called in-place sorting.

Binary Search

```
int binarySearch(int A[ ], int low, int high){

 if(low > high) return -1;

 else {

        mid = (low+high)/2;

        if(A[mid]== key){

        return mid;

        }

        else if(A[mid]> key){

        return binarySearch(key, mid+1, high);

        }

        else{ /*A[mid]> key*/

        return binarySearch(key, low, mid-1);

        }

 }

}
```

Analysis of Binary Search: Recurrence relation for binary search algorithm can be written as following.

$$T(n) = 1 + T(n/2)$$

- **Best Case Analysis:** When the element is in the middle place of the given input array, it searches only one element.
    - Best case Time complexity: $T(n) = O(1)$
- **Worst Case Time Complexity:** $T(n) = O(\log n)$
- **Average Case Time Complexity:** $T(n) = O(\log n)$

Greedy Design Technique

- Greedy algorithms uses the heuristic of making the locally optimal choice at each stage of problem solving, with the hope of finding a globally optimal.
- An optimization problem can be solved using following Greedy approach.
    - Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.
    - Optimal substructure: An optimal solution to the problem contains an optimal solution to subproblems.

Greedy Algoritms

- Single Source shortest path (Dijkstra's Algorithm)
- Minimum Spanning Tree (Kruskal's algorithm & Prim's algorithm)
- Huffman Coding (Optimal prefix code)
- Fractional knapsack problem
- Activity Selection Problem
- Optimal Merge Pattern

Huffman Coding

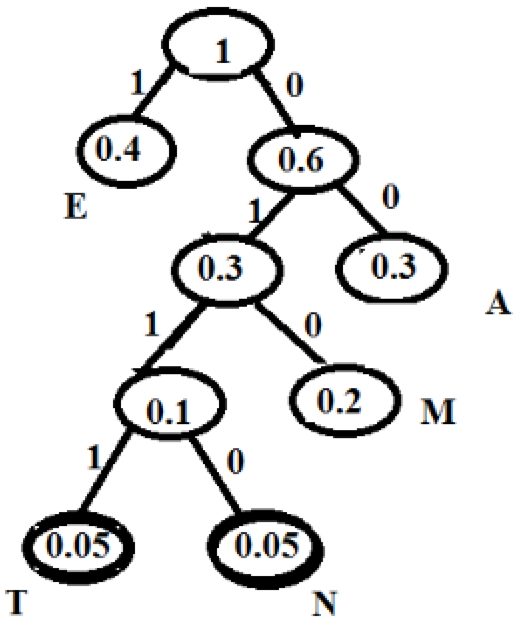
- Storage space for files can be saved by compressing them where each symbol can be replaced by a unique binary string.
- Codewords can differ in length.
- Each code need to be prefix free that is no codeword is a prefix of another code.
- The Huffman coding is a greedy method for obtaining an optimal prefix-free binary code.

Example: Find the average word length for the following letters with the given frequency of use of the letters.

E – 40%; A – 30%; M-20%; N-5%; T-5%

Solution:

Following huffman tree can be constructed using the given frequencies.



Letter	*Code*	*Length*	*Frequency*	*Length × frequency proportion*
E	1	1	0.4	0.4
A	00	2	0.3	0.6
M	010	3	0.2	0.6
N	0110	4	0.05	0.2
T	0111	4	0.05	0.2
			Total	$\boxed{2.0}$

Average word length is 2.

Dynamic Programming

Dynamic programming helps to solve a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions.

A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem.

Dynamic Programming Algorithms

- Fibonacci sequence
- 0-1 knapsack problem
- Maximum subarray problem
- Longest Common Subsequence problem
- Matrix chain multiplication

- All pairs shortest path problem

Minimum Spanning Trees

A **spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.

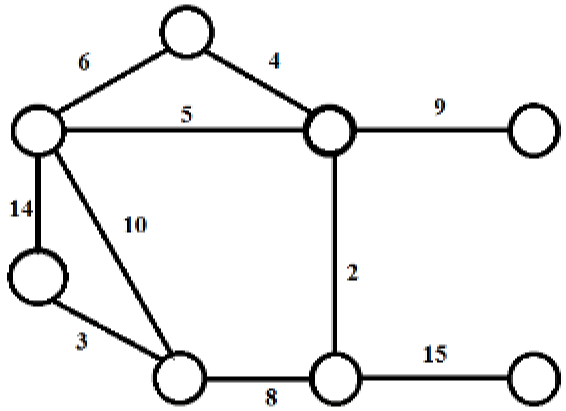
Let a Graph G has n vertices and e edges.

- Then Spanning tree T of Graph G contains n vertices and (n-1) edges.
- Spanning tree T has no cycles which is a tree.

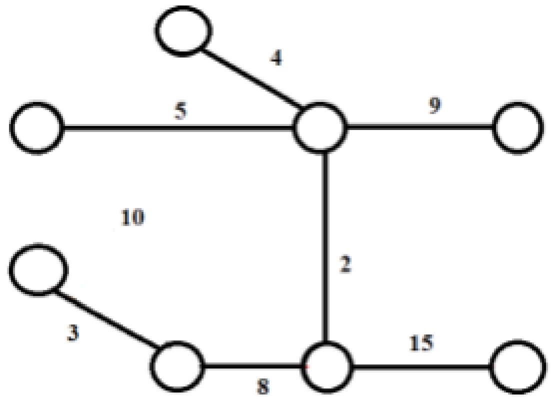
Minimum Weight Spanning Tree (MST): It is a spanning tree where the total edge weight is minimal among all the possible spanning trees of the given graph G. Such spanning tree with minimum weight is called minimum weight spanning tree (MST).

- An MST is not necessarily unique.

Example: Consider the following graph G. Find the minimum weight spanning tree of G.



Spanning tree for the above graph G is:



There are two popular Greedy algorithms for finding the minimum spanning tree.

- Kruskal's Algorithm
- Prim's Algorithm

Prim's Algorithm computes MST by starting at one vertex and grow the tree.

Kruskal's Algorithm

- It follows greedy approach to compute the minimum spanning tree.
- Kruskal's Algorithm computes minimum spanning tree by considering the edges in increasing order of weight.
- Intermediate result of kruskal's algorithm may be connected or disconnected graph.
- It can be implemented using Union find data structure.

Pseudo code for Kruskal's algorithm:

1. Sort all edges in increasing weight order.
2. Select an edge with minimum weight.
3. If the edge does not create a cycle, add it to the spanning tree. Otherwise discard it.
4. Stop, when n − 1 edges have been added. Otherwise repeat step 2 to step 3.

- **MAKE-SET(v) puts v in a set by itself**
- **FIND-SET(v) returns the name of v's set**
- **UNION(u, v) combines the sets that u and v are in**

MST-Kruskal(G, w)

```
A ← ∅
for each vertex v ∈ V[G]
      do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
      do if FIND-SET(u) ≠ FIND-SET(v)
            then A ← A ∪ {(u, v)}
                  UNION(u, v)
return A
```

<u>Explanation</u>

Let $G = (V, E)$ be the given graph, with $|V| = n$

{

Start with a graph $T = (V, \phi)$ consisting of only the vertices of G and no edges; /* This can be viewed as n connected components, each vertex being one connected component */

Arrange E in the order of increasing costs;

for ($i = 1, i \leq n - 1, i + +$)

{ Select the next smallest cost edge;

if (the edge connects two different connected components)

add the edge to *T*;

}

}

Analysis of Kruskal's algorithm:

- Step 1: Initialization of Array takes O(1)
- Step 2 to Step 3: First for loop takes |V| MAKE-SETs
- Step 4: Sorting |E| edges takes O(|E| log |E|)
- Step 5 to Step 8: Second loop iterates O(|E|) times. Total time here is O(|E| log |V|)

$$|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V).$$

Total time of Kruskal's Algorithm: O(|E| log |E|) = O(|E| log |V|)

- Note: If edges are already sorted: this algorithm will take linear time.

Prim's algorithm

- It can be implemented using priority queue.
- Intermediate result of prim's algorithm always connected graph.

Pseudo code for Prim's algorithm:

We keep a priority queue filled with all the nodes that have not yet been spanned. The*value* of each of these nodes is equal to the smallest weight of the edges that connect it to a the partial spanning tree.

1. Initialize the Priority queue with all the nodes and set their key values to a number larger than any edge, and the parent of the root to nil. Set the key value of the root to 0. **(Step 1 to 5)**
2. While Priority queue is not empty do { Let u be the minimum value node in the queue; For every node v in u's adjacency list do { if v is in queue and the weight on the edge (u,v) is less than key value(v) { Set parent of v to u; Set key value(v) to weight of (u,v); } } } **(Step 7 to 11)**

- INSERT(v) puts v in the structure
- EXTRACT-MIN() finds and returns the node with minimum key value
- DECREASE-KEY(v, w) updates (decreases) the key of v

```
MST-Prim(G, w, r)
1   for each u ∈ V[G]
2         do key[u] ← ∞
3            π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ ∅
7         do u ← EXTRACT-MIN(Q)
8            for each v ∈ Adj[u]
9                 do if v ∈ Q and w(u, v) < key[v]
10                      then π[v] ← u
11                           key[v] ← w(u, v)
```

Analysis of Prim’s Algorithm

- **Step 1 to Step 3:** Takes O(|V|) time.
- **Step 6 to Step 11:** Takes O(|V| log |V|) + O(|E| log |V|); Here Extract Minimum operation at step 7 takes O(log |V|), and Updating queue (Decrease key operation) from step 9 to step 11 also takes log |V| time using binary heap.
- Time complexity for Prim’s algorithm (using binary heap as above) is **O(|E| log |V|)**.
- If Prim’s algorithm implemented using fibonacci heap, then time complexity is**O(|E|+ |V| log |V|)**.
- If Prim’s algorithm implemented using adjacency list, then time complexity is O($|V|^2$).

Shortest Paths

Single Source Shortest Paths

Shortest path problem is to determine one or more shortest path between a source vertex s and a target vertex t, where a set of edges are given.

Consider a directed graph G = (V, E) with non-negative edge weight and a distinguished source vertex, s ∈ V. The problem is to determine the distance from the source vertex to every other vertex in the graph.

Dijkstra’s Algorithm

- Dijksta's algorithm solves the single source shortest path problem on a weighted directed graph.
- It is Greedy algorithm.
- Dijkstra's algorithm starts at the source vertex, it grows a tree T, that spans all vertices reachable from S.
- Dijkstra's algorithm works similar to Prim's algorithm.
- It uses BFS (Breadth First Search) to find the shortest distances.
- The following simple algorithm explains the working principle of Dijkstra's algorithm.

```
let T be a single vertex s;
while (T has fewer than n vertices)
{
   find edge (x,y)
      with x in T and y not in T
      minimizing d(s,x)+length(x,y)
   add (x,y) to T;
   d(s,y)=d(s,x)+length(x,y);
}
```

Dijkstra's Algorithm Implementation :

```
function Dijkstra(Graph, source):

      dist[source]  := 0                      // Distance from source to source
      for each vertex v in Graph:             // Initializations
          if v ≠ source
              dist[v]  := infinity            // Unknown distance function from source to v
              previous[v]  := undefined       // Previous node in optimal path from source
          end if
          add v to Q                          // All nodes initially in Q
      end for

     while Q is not empty:                    // The main loop
         u := vertex in Q with min dist[u]  // Source node in first case
         remove u from Q

         for each neighbor v of u:            // where v has not yet been removed from Q.
             alt := dist[u] + length(u, v)
             if alt < dist[v]:                // A shorter path to v has been found
                 dist[v]  := alt
                 previous[v]  := u
             end if
         end for
     end while
     return dist[], previous[]
 end function
```

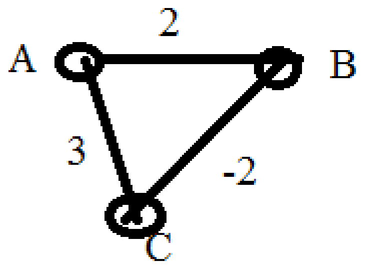
Analysis of Dijkstra's Algorithm:

Time complexity is similar to Prim's algorithm

- **Using Heap :** Time complexity is O(|E| log |V|)

- **Using Fibonacci Heap :** Time complexity is O(|E| + |V| log |V|)
- **Using Adjacency list :** Time complexity is $O(|V|^2)$

Note: Dijkstra's algorithm does not work with negative edge weights. It is important to know that Dijkstra's algorithm requires that weights of all edges are non-negative. Otherwise the procedure is not able to determine whether the shortest path for the node was already found or not.



Let A be the starting (source) vertex in the above graph. Then shortest distance from A will be selected as 2. If we want to find shortest distance from A to B, it will result 2 using Dijkstra's algorithm. But actual shortest distance is 1 if we choose A to C to B path.

Bellman Ford Algorithm

- Bellman-Ford algorithm solves the single-source shortest-path problem in the general case in which edges of a given digraph can have negative weight as long as given graph G contains no negative cycles.
- Bellman Ford not uses Greedy approach to find the shortest paths.
- The algorithm requires that the graph does not contain any cycles of negative length, but if it does, the algorithm is able to detect it.

Pseudo Code for Bellman Ford:

```
bellmanFord(G, s)

    for all edges in G(V)
        D(V) = INT_MAX
        parent[V] = -1

    D(s) = 0

    for i = 1 to |G(V)| - 1
        for each edge (u, v) in G(E)
            if edge can be Relaxed
                D(v) = D(u) + weight of edge (u, v)
                parent[v] = u

    for each edge in G(E)
        if edge can be Relaxed
            return false

    return true
```

Analysis of Bellman Ford Algorithm:

We assume that the algorithm is run on a graph G with |*V*| nodes and |*E*| edges. Let |V|=n, and |E|=m.

1. At the beginning, the value ∞ is assigned to each node.This requires O(n)
2. Then we do the *n-1* phases of the algorithm: one phase less than the number of nodes. In each phase, all edges of the graph are checked, and the distance value of the target node may be changed. We can interpret this check and assignment of a new value as one step and therefore have *m* steps in each phase. In total all phases together require*m* · *(n-1)* steps. This requires O(mn)
3. Afterwards, the algorithm checks whether there is a negative circle, for which he looks at each edge once. Altogether he needs *m* steps for the check. O(m)

Total Time complexity for Bellman Ford Algorithm: **O(m n) = O(|E|.|V|)**

Floyd – Warshall Algorithm

- It can find **shortest (longest) paths** among all pairs of nodes in a graph, which does not contain any cycles of negative length.
- It is dynamic programming algorithm.
- It can be used to detect the presence of negative cycles.
- It can find transitive closure for directed graph.

Pseudo Code for Floyd-warshall algorithm:

Let n be the number of vertices in the graph G.

- Pred [x,y] can be used to store the reachability and will help to extract the final path between two vertices.
- d[n,n] will store the result for all pairs shortest paths.
- w[i, j] contains the weight of an edge (i, j) for the given graph G.

```
Floyd-Warshall (w, n)
{
    for (i=1 to n) {
          for (j=1 to n) {
          {
                d[i, j] = w[i,j];
                Pred[i, j]=NIL;
          }
    }
                                                    1
    for (k=1 to n) {
          for (i=1 to n) {
                for (j=1 to n) {
                      if( (d[i, k] + d[k, j]) < d[i, j] ) {
                            d[i, j] = d[i, k] + d[k, j];
                            Pred[i, j] = k;
                      }
                }
          }
    } //d is updated with all pairs shortest path
                                                    2
}
```

```
void FloydWarshall(int distance[maxVertices][maxVertices],int vertices)
{
        int from,to,via;
        for(from=0;from<vertices;from++)
        {
                for(to=0;to<vertices;to++)
                {
                        for(via=0;via<vertices;via++)
                        {
                                distance[from][to] =
min(distance[from][to],

 distance[from][via]+distance[via][to]);
                        }
                }
        }
}
```

Analysis of Floyd – Warshall Algorithm:

1. In the beginning of code there are two loops, which requires O(n^2) time and two statements inside the inner loop takes constant time.
2. Next, there are three loops. These three loops can require O(n^3) time. If statement of the inner loop requires constant time.

Time complexity of Floyd – Warshall Algorithm = **O(n^3)**